Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Miroslav Chomut

Tool for editing PDDL projects

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Tomáš Plch

Study programme: Computer Science

Specialization: Programming

Prague 2012

I would like to thank my supervisor Mgr. Tomáš Plch, who made this work possible. I would like also to thank Google Company for their great search tool which helped me a lot. Last but not least I would like to thank to all the people who helped me with this work in any way.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date.....

Názov práce: Nástroj na editáciu PDDL projektov

Autor: Miroslav Chomut

Katedra / Ústav: Kabinet software a výuky informatiky

Vedúci bakalárskej práce: Mgr. Tomáš Plch

Abstrakt: Planning Domain Definition Language (PDDL) je jeden zo štandardných jazykov, ktoré sa používajú na definovanie plánovacích domén a problémov. PDDL je syntakticky komplikovaný jazyk, a preto sa vývojári často dopúšťajú syntaktických a sémantických chýb. Práca s rozsiahlymi PDDL súbormi je náročná. Na rozdiel od imperatívnych programovacích jazykov (napr. C#, C++), pre PDDL neexistuje vhodný rozšírený editor.

Našim cieľom je poskytnúť vývojárom nástroj na komfortnú editáciu PDDL projektov, ktorá je známa z nástrojov pre imperatívne programovacie jazyky (napr. Microsoft Visual Studio). Táto práca popisuje náš nástroj, ktorý nazývame PDDL Studio a podporuje a) zvýrazňovanie kódu, b) kontextovú kompletáciu kódu, c) detekciu chýb s interaktívnou tabuľkou chýb, d) kolapsáciu kódu, e) projekt manažment, f) XML export a import, g) integráciu plánovača a h) bežné funkcie editorov (napr. párovanie zátvoriek, číslovanie riadkov). Náš nástroj je multiplatformový a je navrhnutý na zvýšenie efektivity PDDL vývojárov a sprehľadnenie kódu.

Kľúčové slová: Problem Domain Definition Language, Integrated Development Environment, editor, syntaktická kontrola, syntaktické podfarbovanie, kompletácia kódu, kolapsácia kódu

Title: Tool for editing PDDL projects

Author: Miroslav Chomut

Department / Institute: Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Tomáš Plch

Abstract: The Planning Domain Definition Language (PDDL) is one of standard languages used for defining planning domains and problems. PDDL is a syntactically complex language therefore developers often make syntax and semantic errors. Working with larger PDDL files is time consuming. Unlike imperative programming languages (e.g. C#, C++), there is no suitable widespread tool for editing PDDL.

Our goal is to provide PDDL developers with tool for comfortable editing, which is known from tools for imperative programming languages (e.g. Microsoft Visual Studio). This thesis describes our project called PDDL Studio, which is capable of a) syntax highlighting, b) context sensitive code completion, c) error detection with Interactive Error Table, d) code collapsing, e) project management, f) XML export and import features, g) planner integration and h) common editor features (e.g. Bracket Matching, Line Counter). Our tool is multiplatform and it is designed to increase efficiency of PDDL developers and code readability.

Keywords: Problem Domain Definition Language, Integrated Development Environment, editor, syntax checking, syntax highlighting, code completion, code collapsing

Contents

Introduction

Planning [3] is a branch of Artificial Intelligence (AI) concerned with creation of action sequences based on environment description and task specification. Building blocks of planning are: domain and problem specification, planner, plan and problem solution. Planning domain describes the environment and its states, relations, available actions, their preconditions and effects as well as restrictions. Example of a planning domain is an airplanes construction. Available actions are moving and connecting components. Possible restriction is the capability of the assembly line. Precondition of component connection is the availability of both connecting components during the process (e.g. welding requires the welding machine).

Planning problem contains an initial state and a goal state description as well as intermediate states, which must be satisfied before satisfying the goal state or during the whole execution of actions. In our example, initial state is a set of components around construction facilities and final state is an airplane. Plan represents sequence of applicable actions respecting a domain, where plan solving a problem is called a *problem solution*. Actions in a problem solution transform the initial state to the goal state, while satisfying intermediate states of the problem. In the airplane example the solution is a set of relocating and connecting actions resulting in successful plane construction. The planner (e.g. SHOP [20], JSHOP2 [21], PLPLAN [18], blackbox [5]) is a tool which takes a domain and a problem specification as input and returns a solution as output. Solving the problem can be viewed as a search through state space of possible plans.

For the purpose of writing down planning problems and domains few languages exist. For example Action Description Language [2] and Problem Domain Definition Language (PDDL) [17]. The PDDL is nowadays commonly used standard for uniform description of planning domains and problems. It is a Lisp [14] based language with prefix notation and was originally invented to be used at competitions and conferences like ICAPS [11] and ICKEPS [12].

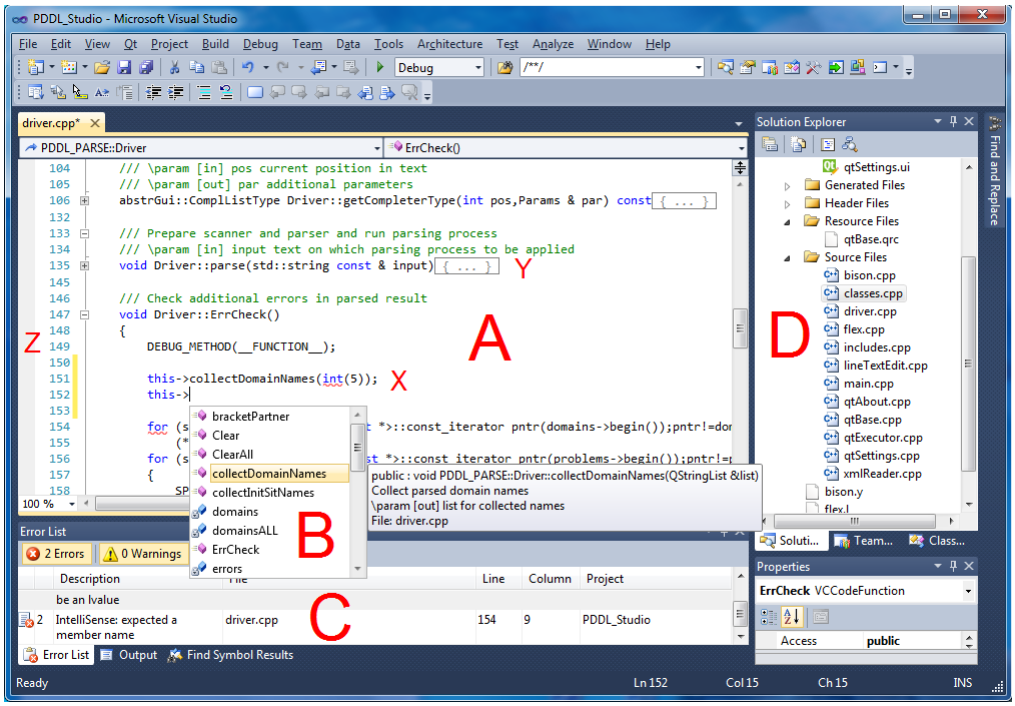
As a part of competitions, contestants are presented with domains written in PDDL. These can be a result of by–hand creation (i.e. in a text editing tool like Notepad++ [15]) or they can be created by tools (e.g. itSimple [10], GIPO IV [22], ViTAPlan [26]) where developer visualizes the domain in a graphical environment. In either case, a developer may need to inspect, analyze and modify domain files by–hand. We recognize a lack of a tool suitable for inspecting, analyzing and modifying PDDL files.

Motivation

From our experience, programmers working with imperative programming languages [13] (e.g. C#, C++) spend most time editing source code. These languages are edited by tools which concepts are sophisticated as a result of decades of development and usage. Examples of such tools are Microsoft Visual Studio [25], Eclipse [8] and NetBeans [27].

Therefore our motivation is to provide PDDL developers with the coding culture and concepts known from tools for imperative programming languages. Our inspiration is Microsoft Visual Studio (Pic 1), most notably these features: a) project management, b) syntax highlighting, c) code collapsing, d) interactive error detection and e) context aware code completion mechanics.

Therefore our goal is to provide planning community with a suitable tool for editing PDDL project comparable with tools well known from imperative programming languages.



Pic 1.1 – Microsoft Visual Studio example where can be seen main editing window (A) with highlighted text, underlined error (X), collapsed code (Y) and line counter (Z). In main window, code completion pop–up (B) can be seen as well. Below main window is error table (C) and to the right is project management (D).

The results of the thesis have been accepted to be published at the ICAPS 2012 conference (International Conference on Automated Planning and Scheduling) as a system demonstration paper.

Structure

In the first chapter we analyze and propose features of our tool, PDDL Studio. In the second chapter we present behavior of proposed features. In the third chapter we describe internal structures of our application. In the fourth chapter we go through implementation details. Last chapter is the conclusion and future work. User guide is in Appendix B.

1 Analysis

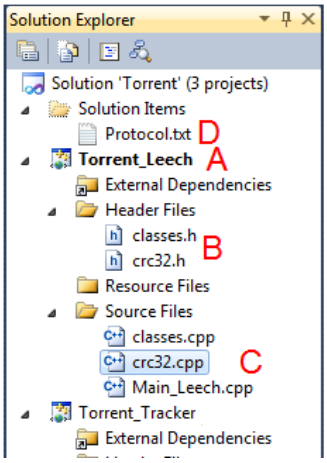
This chapter describes our analysis based on our inspiration. Here we analyze the features chosen from Microsoft Visual Studio and we describe their usage and propose appropriate form in our tool.
We chose the following features:

- Project Management – encapsulates project files, keeps track of modified status and error count
- Syntax Highlighting – colorization of text elements
- Code Collapsing – temporarily shrinks unneeded block of text
- Syntax and semantic Error Detection with Interactive Error Table – detects errors and collects them in table
- Context aware Code Completion – predicts typed word and can finish it
- XML export and import – exports PDDL file into our XML format and vice versa
- Integration of third party software – ability to utilize third party software from within the application
- Common editor features – Line Counter, Bracket Matching, Auto–Save
- Logging – provision of program runtime information for future upgrades

1.1 Project Management

An application's Project usually consists of multiple files, which can be code files (e.g. hpp, cpp) and resource files. Project Management bounds these files together and manages them. For example when programmer opens a Project, the Project Management loads all important files. When a programmer compiles a Project, the Project Management ensures that all code files are saved and runs the compiler.

Good example of Project Management is the one in Microsoft Visual Studio. Main entity of Microsoft Visual Studio's Project Management is called Solution, which aggregates application Projects and their code and resource files as well. The Solution description file contains information about all files. Picture (Pic 1.1) shows Microsoft Visual Studio's Project Management interface with loaded Projects consisting of misc file (Pic 1.1 (D)) and multiple applications (Pic 1.1 (A)) each with its header files (Pic 1.1 (B)) and source files (Pic 1.1 (C)).



Pic 1.1 – Visual Studio Project Management demonstrating multiple applications (A) with their header files (B) and code files (C). Project also contains misc file (D).

Project Management is nowadays a standard feature of many tools. It simplifies programmer's work by relieving routine management tasks. A programmer concentrates on the productive part of the development and others are handled by Project Management. Project Management also supports other features working with whole project (e.g. interfile error detection and code completion).

In our case we need Project Management to encapsulate PDDL source files, provide information about them to other features (e.g. error detection, code completion), provide the developer with status of individual files (e.g. error, modified and not saved) and handle project file management (e.g. add new file, delete file, save all files).

1.2 Syntax Highlighting

Colorization of code elements based on syntax is called Syntax Highlighting [23]. Syntax Highlighting is proven as very helpful feature for structured languages. Picture (Pic 1.2) shows an example of highlighted C++ code in Microsoft Visual Studio. Syntax Highlighting improves readability of the source code by distinguishing different elements (e.g. keyword, number, text). The developer can quickly identify parts of the code without need to read all the code around.

```
/// Structure containing information about one error
struct ErrorType {
    posType pos;              ///< Error starting position
    std::string const text;   ///< Error message
    std::string const origin; ///< Error origin

    ErrorType(posType const pos_,std::string const & text_);  ///< Constructor
    void Process(abstrGui & gui);                             ///< Highlights
    ~ErrorType();                                             ///< Destructor
};
```

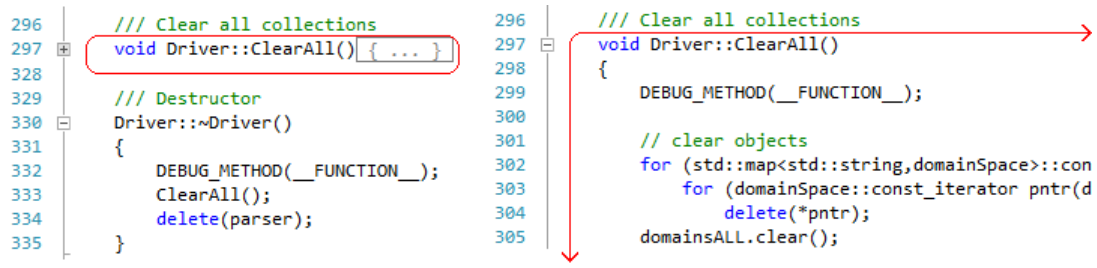
Pic 1.2 – Microsoft Visual Studio example of highlighted code

PDDL is a structured language and consists of distinct language elements on which highlighting can be applied. Therefore we decided to include Syntax Highlighting into our tool. For Syntax Highlighting to work the code must be parsed first, therefore we need to implement a parser for PDDL.

1.3 Code Collapsing

Source code of imperative programming languages can be differentiated into distinguished code blocks. Many source code editors make use of this and provide programmer with Code Collapsing feature which can temporarily hide unneeded code blocks. Example of Code Collapsing in Microsoft Visual Studio can be seen in picture (Pic 1.4), where `ClearAll()` method of `Driver()` class is collapsed.

This feature improves inspecting of the source code, as programmer can easily collapse unneeded blocks.

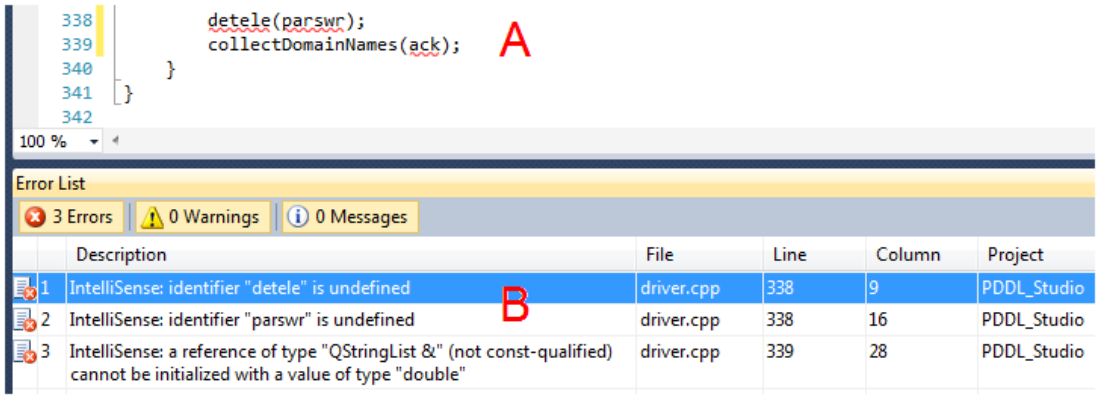


Pic 1.3 – Microsoft Visual Studio example of collapsed (left) and expanded (right) code

In PDDL code blocks can be recognized too, therefore implementing Code Collapsing feature capable of collapsing bigger outer blocks is possible. To detect code blocks borders the code must be parsed first, therefore we need to implement a parser for PDDL.

1.4 Error Detection

Nowadays most of tools for imperative programming languages have implemented some interactive Error Detection. Error Detection often detects syntax errors and some better tools are able to detect semantic errors as well. Example of Error Detection in Microsoft Visual Studio can be seen in picture (Pic 1.5), where errors are underlined in code and are displayed in an error table.



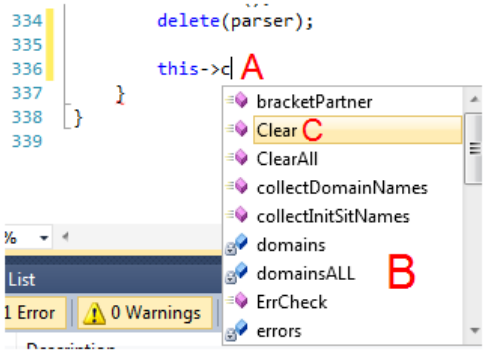
Pic 1.4 – Microsoft Visual Studio Error Detection where errors are underlined (A) and collected in error table (B)

Benefit of the interactive Error Detection is the fact that the programmer is aware of the error immediately after making one. In case of misspell keywords and simple errors, programmers can quickly repair them.

The PDDL grammar contains keywords which can be guarded for syntax errors and system of structures which can be guarded for semantic errors. We decided for implementing error detection into our tool to enhance developer's effectiveness. Evaluation of errors takes place on parsed code rather than on unparsed one, therefore we need to implement a parser for PDDL.

1.5 Code Completion

Many tools for imperative programming languages support Code Completion. This feature allows programmers to write only first few characters of desired keyword and a pop–up menu provides possible completions fitting into current context. Programmer chose and confirms desired keyword and Code Completion completes the keyword. Example of code completion in Microsoft Visual Studio can be seen in picture (Pic 1.6), programmer typed "`this->c`" and pop–up provided him/her with possible completions.



Pic 1.5 – Microsoft Visual Studio code completion pop–up example, programmer typed part of keyword (A) and code completion pop–up (B) appeared, programmer chose desired keyword (C)

Code Completion has many benefits. With an appropriate naming convention, programmers do not need to go through the code or documentation because Code Completion provides them with appropriate hints.

In the PDDL grammar, when a part of an element is finished, some keywords can be guessed and parameters of predicates are limited to predefined ones. So there is a place for Code Completion feature in our tool. As with Error Detection, Code Completion utilizes parsed code to work, so we need a parser for PDDL.

1.6 XML export/import

Extended Markup Language (XML) [29][28] is popular and widespread format, while PDDL is hard to read and parse. By exporting PDDL file into XML, we allow developers unfamiliar with PDDL to work with PDDL files. Therefore we decided to implement an XML export and import feature.

1.7 Integration of third party software

During development of bigger programs, it is common that part of the source code is pre–generated (e.g. by Flex [9], by Bison [4]), major part of code editing is done in a code editor and the application compiled by a compiler. The most of these tools are external in respect to the editor.

Therefore many tools provide integration of third party tools as standard feature. For example when a programmer wants to compile a program, Project Management ensures that all important files are saved, calls external pre–processors if needed (e.g. Flex, Bison) and finally calls a compiler. Full action is controlled by the programmer from within the source editor. Benefit is a modular, all under one roof, approach with ability to adapt to new third party tools.

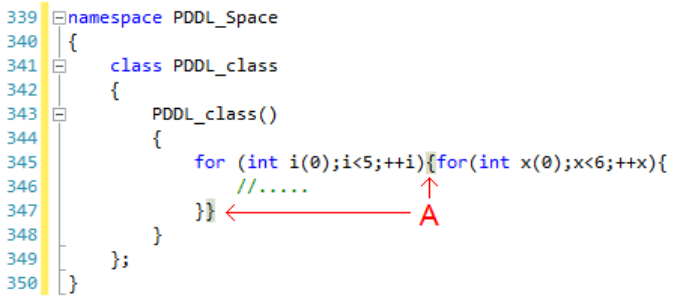
As programs are being compiled, PDDL projects are being interpreted by PDDL planners. In the same way as compilers are enhanced and developed by time

the same stands for PDDL planners. Integration of third party software makes changing of planners possible as they evolve without modifying our tool.

1.8 Common editor features

Line Counter feature improves readability and is used by almost all tools for manual text editing. Line Counter allows programmers to relocate faster while scrolling between distant parts in the source code.

Bracket Matching feature improves orientation in structured code by highlighting of current bracket pair. It allows programmers to quickly recognize block's borders or wrong brackets count. Picture (Pic 1.3) shows example of Line Counter in Microsoft Visual Studio along with Bracket Matching feature.



Pic 1.6 – Microsoft Visual Studio example demonstrating Line Counter and Bracket Matching feature (A)

Auto–Save feature periodically saves unsaved work. Thanks to Auto–Save feature, when crash occurs, unsaved work can be salvaged from automatically saved files.

PDDL language is Lisp [14] based and contains a multitude of nested logical blocks enclosed in brackets. Example (Code 1) shows possible accumulation of closing brackets at one position. Therefore both features, Line Counter and Bracket Matching, are viable adepts to be implemented into our tool. Auto–Save feature might prove useful as well.

```
(...(...
 :effect
      (and (holding ?x)
      ...
      (not (on ?x ?y)))))
```

Code 1 – Example code showing wide use of brackets

1.9 Logging feature

During beta testing and deployment, bugs can occur: unexpected platform or Operating System's behavior, malformed input data etc. These unexpected situations can result in an application crash.

When an application crashes, runtime information is lost therefore recreating the pre–crash conditions can be a complicated task. Applications solve this by providing Runtime Logging feature. When the application crashes the log can be

used by programmers for debugging. We intend to implement the Logging feature in our tool for exactly this purpose.

1.10 Multiplatform approach

The planning community is widespread and so are their tools. Some planners run on the Microsoft's operating systems platform, while others prefer a UNIX based platform. We decided to implement our tool multiplatform so we can support wide part of the community. Because our tool is an editor in nature, we need to utilize a GUI framework or implement the GUI itself with respect to maintaining a multiplatform approach.

1.11 Parser

PDDL is structured language and many features utilize its characteristics (e.g. Error Detection, Code Completion) therefore we need to implement a parser. We can create one parser for each feature or we can create one parser suiting all parser dependent features. Because all parser dependent features are expected to run simultaneously, one general parser suiting all of them is a better approach.

2 Design

In this chapter we describe the behavior of features proposed in our analysis. We progress in the following order:

- PDDL Parser – parses PDDL into tree–like structure, which can be utilized by application
- Project Management – encapsulates project files, keeps track of modified status and error count
- Syntax Highlighting – colorizes text elements
- Code Collapsing – temporarily shrinks unneeded block of text
- Syntax and Semantic Error Detection – detects errors and collects them in table
- Code Completion – predicts keywords based on the current context
- XML export and import – exports PDDL file into our XML format and vice versa
- Integration of third party software – allows utilization of a third party software (e.g. planner) from within the application
- Common editor features – provides Line Counter, Bracket Matching, Auto–Save
- Runtime Logging – provides program runtime information for runtime logs (i.e. for debugging)

2.1 PDDL Parser

PDDL is structured language with tree–like structure and many features of our tool utilize its characteristics:

- Syntax Highlighting requires the ability to differentiate colorized elements.
- Code Collapsing needs to recognize collapsible block borders.
- Error Detection needs to detect syntax (e.g. misspelled or missing keyword) and semantic (e.g. inconsistent parameters type for predicate) errors.
- Code Completion generates two types of completion lists: a) static syntactic (e.g. PDDL keywords) and b) dynamic semantic list (e.g. predicate names). Generation of these lists is based on the tree–like structure and position in it.
- XML export uses tree–like structure to produce output in XML format.
- From common editor features, the Bracket Matching requires localization of bracket pairs.

Therefore we need a PDDL Parser, which parses PDDL into a forest of tree–like structures, where one node represents one element of the PDDL and node's child represents nested element of the parent element. Each feature then calls appropriate method on the tree's root. Every node handles a request (i.e. method call) and propagates the call into its children. The PDDL Parser should be standalone capable and usable as state parser for future development and possible use outside of our application. Because the Parser is expected to be used in an interactive environment, its parsing speed is to be considered.

2.2 Project Management

Project Management is a component responsible for handling projects with corresponding PDDL files. We recognize files with "`pddl`" extension as PDDL documents and beyond developer's modifications we handle them without inclusion of any additional content (e.g. XML header). Developer can work with one file at a time, which is called an *active file*.
Project Management has the following tasks:

- Manage projects:
  - Create a project – creates new empty project.
  - Open a project – opens an existing project.
  - Save a project – saves current project with all its files.
- Manage project's files:
  - Add file – adds an existing PDDL file or creates new empty file.
  - Remove a file – after confirmation removes the file from a project, without erasing it from a persistent storage (e.g. a hard drive).
  - Keep track of project's files – persistently keeps track of project's files. To do so it stores relative path to project's files in additional file called the Project File with "`pddl_proj`" extension and provides PDDL developer with list of all project's files.
  - Select active file – active file is the one that can be inspected, analyzed and modified (only one active file at a time is allowed), however multiple project's files can be opened.
  - Save files – persistently stores files onto a persistent storage (e.g. a hard drive).
- Provide developer with file statistical information
  - Modified status – visually differentiates files that contains unsaved modifications.
  - Error count – provides developer with number of detected errors for each project's file.
- Support other features – some features may need information on project's files (e.g. Error Detection utilize list of all files, integration of third party software needs relative path to project's files), therefore obtaining these information must be possible.

2.3 Syntax Highlighting

Syntax Highlighting is a feature that colorizes different elements of a text and PDDL is structured language, which contains both fixed and user defined elements that can be colorized:

- Predicates (a PDDL element) – colorized by "`Predicate`" highlight
- Variables (i.e. variable names) – colorized by "`Variable`" highlight
- Types (e.g. type of variable) – colorized by "`Type`" highlight
- Other user defined symbols (e.g. domain name, problem name) – colorized by "`Name`" highlight
- Keywords – colorized by "`Keyword`" highlight

We include three additional highlights: a) "`Normal`" for un–highlighted text, b) "`Error`" for any errors detected by the Error Detection and c) "`Highlighted Bracket`" for bracket highlighting. Syntax Highlighting

utilizes parsed tree–like structure to determine highlighting, therefore it is parser dependent. Developer should be able to change highlights as well as font of source code editor.

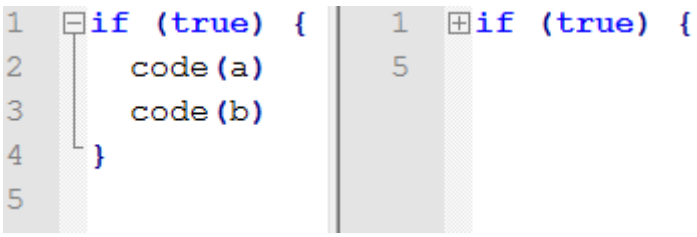
2.4 Code Collapsing

Code Collapsing is a feature, which allows a developer to temporarily hide unneeded lines, respectively collapse larger block into a single line. Code Collapsing is controlled and facilitated by a GUI component (i.e. editing window), which displays and allows modification of the PDDL code.

Because we want to use Code Collapsing on PDDL code blocks, the GUI component needs borders of such blocks to determine where the collapsing is applicable along with information whether certain block is collapsed or not, for correct visualization. Source of the block borders is the parsed tree–like structure therefore we need some matching mechanism to preserve the collapsed state of unchanged blocks when tree–like structure changes due to a change in the PDDL code. When new block borders are retrieved from the parsed tree–like structure, they are matched with actual block borders stored in the GUI component to set collapsed state of matched elements before their replacement.

GUI component is not expected to differentiate between modification in PDDL code and selection of a different active file therefore the Project Management needs to inform the component about change of the active file to make the collapsed state of blocks persistent between project's files. However collapsed states are not saved onto persistent storage, therefore after a project is closed, they are lost.

It is common to position collapse/expand switches on the left edge of the GUI element close to the code (Pic 2.1) therefore we follow this tradition.



Pic 2.1 – Example of Code Collapsing in Notepad++ [15], collapse switch visualized as "-", expand switch visualized as "+".

2.5 Error Detection

Interactive Error Detection is a mechanism that detects errors while developer types. Our application stores errors in an error collection, which is interconnected with Project Management, so errors are collected individually for each project's file. Error collection is filled by three types of errors:

- Syntax errors (e.g. missing keyword) collected from a parsed tree–like structure. Each node of the parsed tree–like structure represents some PDDL element and is capable of self diagnosis – recognizing basic syntax errors like missing keyword, missing inner element, wrong order of inner elements etc.
- Semantic errors (e.g. wrong type of predicate parameters) collected from parsed tree–like structure. Certain nodes of tree–like structure are capable of detecting some semantic errors. For this purpose nodes that have a root different than a domain definition (e.g. nodes of a problem definition) have

a root node of a domain definition it is defined on at its disposal. Examples of detected semantic errors:

  - Certain elements are valid only when allowed in "`requirements`" clause of parent domain definition. Nodes representing these elements verify this and reports error when inconsistency is detected.
  - Nodes that represent use of a predicate verify whether the predicate is defined on the parent domain and all used parameters' types are compatible to the predicate definition.
  - Nodes that stores information about used type, verify whether the type is defined in the parent domain definition.
  - Nodes that stores information about used variable, verify whether the variable is defined in the corresponding area (i.e. is defined in constants, global variables or local variables).

- Errors found during parsing. Some errors cannot be stored into a parsed tree–like structure, either because the erroneous part can't be represented by any node or there is no mechanism to include erroneous node into the tree-like structure at its current state. For example "(define (domain domain foo))" contains duplicity of keyword. These errors must be reported into collection during parsing process, otherwise they would be lost. Examples of such errors reported by the Parser:
  - Misspelled keyword. The structure can carry information about a missing keyword however there is no space for incorrect or abundant keyword.
  - Misplaced whole element, as the element is misplaced there is no mechanism which would place it correctly into the parsed tree-like structure.
  - Duplicity of singleton element or keyword. Each node can carry information about one instance of singleton element, therefore when the Parser asks a parent node to assign singleton child node when it is already assigned, the parent node reports an error as it has no space for storing information about more singleton elements.

Error Detection is expected to be run only on active file when developer makes a modification. However some errors are of interfile nature, for example a developer changes definition of a predicate in one file and its usage in another file becomes corrupt. Therefore we need to run error detection on all files once a while. Collected errors are presented to developer in three ways:

- Most probable error position in source code editor is colorized with special highlight – "`Error`". Most probable position for error is obtained for each error individually. For example: a) misspelled or duplicated keyword is considered as error position itself, b) missing keyword has as error position the preceding keyword or the start of the block, c) mismatching parameter in a predicate has as error position the parameter in the predicate call etc.
- Developer is provided with Interactive Error Table containing all collected errors for active file, capable of moving actual text position onto the selected error on double–click. Table provides developer with error position, error originator and brief error description.
- Each record in project's file list contains information how many errors are detected in the file it represents.

2.6 Code Completion

Code Completion is a feature which provides a developer with contextual hints while he/she types and may complete developer's selection. To acquire the list with matching words we utilize the parsed tree–like structure. Each root node is provided with a cursor position in a PDDL code and the corresponding node is expected to return the list type, which is used for contextual hints. We recognize two list types:

- Static lists. These lists correspond to syntax hints and usually provide developer with possible keywords matching into current position. If this list type is acquired, it is immediately used as it is prepared in advance. Examples of static lists:
  - Keywords for "`requirements`" clause
  - Elements in domain definition
  - Elements in problem definition
  - Elements in action definition
  - Elements in action specification
  - Type of goal description
  - Type of action constraint
  - List containing one keyword that is inevitable
- Dynamic lists. These lists correspond to semantic hints and provide developer with keywords defined earlier in the PDDL code. If this list type is acquired, it must be created prior to its usage. Corresponding root node in tree–like structure is asked to collect the list, which is then used. Examples of dynamic lists:
  - All domain names
  - All initial situation names
  - All predicates of corresponding domain
  - All locally valid variables (including constants)

Code Completion utilizes parsed tree–like structure and the Parser is expected to be run shortly after a PDDL developer pauses the work, therefore is expected to pause the work for a while before utilizing this feature. Code Completion as a pop–up showing right in the edited area is a standard behavior of other editing tools therefore we implement it this way too.

2.7 XML Export/Import

Our tool supports export of individual PDDL files into XML and vice versa. Export utilizes parsed tree–like structure to provide XML output. However during import, we do not have any parsed structure therefore we need XML reader, which creates PDDL output.

2.8 Integration of third party software

Developer may need to run a PDDL document through a planner or multiple planners for performance comparison. Planners are usually command line tools and developer is expected to be familiar with them. Therefore we provide him/her with an ability to prepare command line commands the same way as he/she would enter them into operating system console. We store prepared commands in a list. We also provide a developer with the ability to save all prepared commands into a file with extension "`pddl_exec`" and load them later.

Bigger lists may need some commentaries or executing only certain lines. Character "#" is not present at the beginning of any command in any commonly known operating system, so we use it for commenting the line. It allows developer to both, comment the prepared commands and write the actual comment.

Execution of uncommented orders in the list is serialized, if developer needs parallel execution, it can be forced. For example MS Windows operating systems achieve this by "start " prefix and most UNIX operating systems achieve the same by " &" suffix.

PDDL developer may profit from having variables for file names and their full path. For example when the project is frequently opened from external media, it can have different path on each machine. Another example is running different files of the project through the planner. It is easier to replace only variables than replacing full file names. Therefore we implement these variables:

- Variable representing project directory –"%D%"
- Variables representing project's files relative path – "%0%" for 1st file, "%1%" for 2nd file and so on. Relative paths are provided by project management.

2.9 Common editor features

Our application has implemented few common editor features: a) Line Counter, b) Bracket Matching and c) Auto–Save.

Showing developer the actual position in edited text may prove helpful and is standard feature of most text editors. Line Counter usually consists of number of line visualized next to the text line. We decided for visualizing the line number to the left of the text area. Moreover our application utilizes status bar for more detailed information about the position: character number from the beginning of the file, column position and line number.

Bracket Matching feature helps developer to localize paired brackets. To find paired brackets we utilize the parsed tree–like structure. When developer moves text cursor on any bracket, the one and its pair bracket are highlighted by "Highlighted Bracket" highlight made for this purpose.

Auto–Save feature is common in many tools, not only text editors. It protects developer's work in case of any crash by periodically saving the work or making a backup (saving to backup file). We decided for backup file, as developer may want to revert to original file and we backup only those files that are modified (i.e. the backup file differs from the original unsaved one). Therefore we can easily check whether application previously crashed or not. If Auto–Save files are found, our application loads its content as current (unsaved) content of the file. We provide developer with ability to set appropriate Auto–Save interval to match project circumstances (e.g. project importance, environment stability, persistent storage speed).

2.10 Runtime Logging feature

For debugging and future development we need Runtime Logging. We decided to log function calls. To log function calls, we need information about entrance and exit of each function. Because functions can exit in many places in the code we decided for Helper object approach. We create extra object at the beginning of each logged function or method. The object logs its creation and its destruction along with

function name (Code 2). For readability enhancement the class presenting the helper object should count them and indent log messages.

```
foo{
      DEBUG_METHOD(__FUNCTION__);   creation of a Helper object
      if (condition)
           throw;                   destruction of the Helper object
      else
           return;                  destruction of the Helper object
}
```

Code 2 – Example code showing usage of a Helper object, "DEBUG_METHOD" macro creates a Helper object which logs its creation and when the function returns by any mean the object logs its destruction.

Because our application is expected to contain many short functions, which calls each other, multitude of nested functions can accumulate and the Runtime Logging can considerably slow the application down even in a disabled state (because at least one clause has to be evaluated per function). Therefore we decided for compile time evaluated macro, which controls the Runtime Logging. This way the Runtime Logging bears absolutely zero performance hit on un–logged application. On the other hand, we need two applications: one with Runtime Logging and one without.

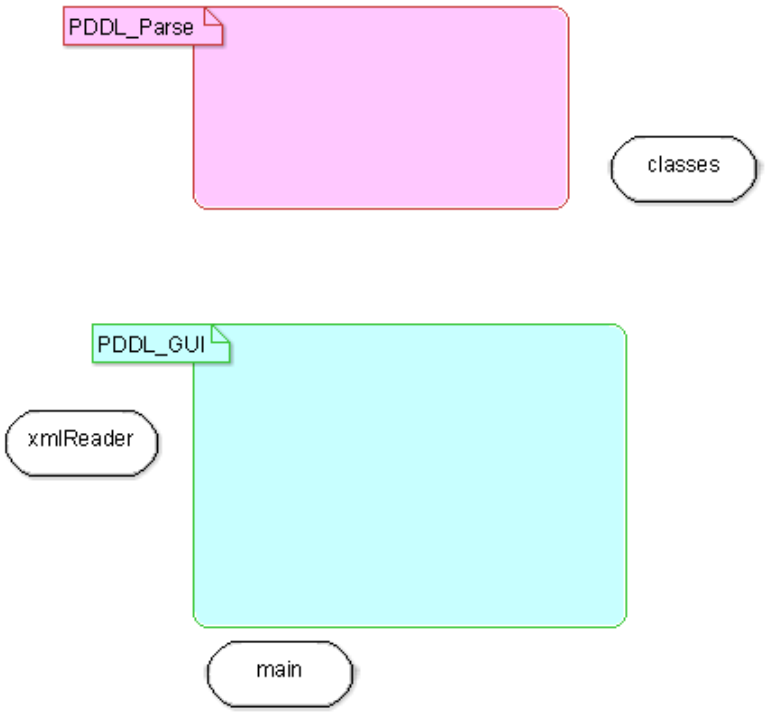
3 Architecture

In this chapter we describe architecture of our tool based on design described in the previous chapter. Our architecture consists of modules. Modules with similar purpose form a block. The modules in a block share a programming language (C++) namespace.

Basically we have two types of features: 1) features working with parsed forest of tree–like structures based on PDDL norm [17], 2) features independent of the parsing process and related to Graphical User Interface (GUI). Therefore we propose few modules and two major blocks:

- Module "`main`" – initializes the program environment
- Module "`classes`" – contains class definitions for each element of PDDL from which parsed forest of tree–like structures is built
- Module "`xmlReader`" – creates PDDL from XML
- Block "`PDDL_Parse`" – contains parser and related tasks
- Block "`PDDL_GUI`" – contains GUI and related parts of features

Basic scheme of major blocks and modules can be seen on picture (Pic 3.1)



Pic 3.1 – Basic schema of architecture: "PDDL_Parse" and "PDDL_GUI" blocks with "classes", "xmlReader" and "main" modules.

3.1 Module "main"

We chose C++ [6] as a platform language because our tool is partly an interactive parser where we expect the need of the full control over program low level behavior for possible tweaks. Initialization of a program environment as well as handling of possible configuration file is implemented in the "`main`" module, so it is separated from the rest of the program.

3.2 Module "classes"

Module "classes" contains class definition for each element of PDDL. Each node in the parsed forest of tree–like structures is instantiation of some from these classes. Most Parser dependent features or at least their parts are implemented as member methods of classes representing nodes (e.g. XML export). Each feature makes call to all root nodes of the parsed forest where each node is expected to call a corresponding method on its child nodes and return the result. For example when a node's method for XML export is called, it creates its XML representation, utilizing appropriate method on its children for their XML representation and returns the XML representation of whole sub tree. Nodes should be capable of following tasks:

- XML Export – each node creates XML representation of its sub tree.
- Syntax Highlighting – each node registers highlights for each highlighted keyword in its sub tree utilizing colorizing method of the GUI.
- Bracket Matching – each node determines whether current bracket pair is included in its sub tree and if yes, it reports positions of the pair.
- Code Collapsing – each node returns block borders of blocks located in its sub tree.
- Error Detection – each node checks its structure for possible errors and reports them.
- Code Completion – each node determines whether current cursor position is within its sub tree and if yes, determines the list type for Code Completion and, in case of dynamic lists, constructs the list.

3.3 Module "xmlReader"

During import operation, we have unparsed XML file and we need to transfer it into PDDL file. Because this feature is not Parser dependent nor does it depend on GUI, we decided for sole "xmlReader" module, which creates a PDDL output from an XML file. The module reads XML file token by token and transfers each token into final PDDL file by forward only approach. XML export can be easily implemented into "classes" module, as it can utilize parsed forest of tree–like structures and each PDDL element can transfer itself into XML.

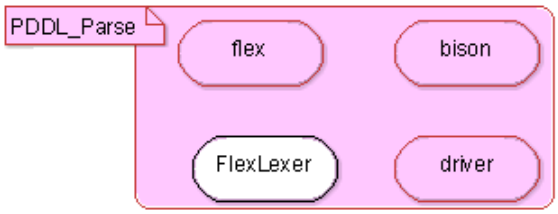
3.4 Block "PDDL_Parse"

Writing parser engine from scratch is a difficult task so we decided to utilize a combination of Flex [9] and Bison [4] to create a parser from a grammar specification. We chose these tools which are mainly designed for C applications and can be easily used with our C++ application.

Flex generates a lexical analyzer providing stream of tokens as its output. Bison generates Parser's core, which utilizes stream of tokens from lexical analyzer and constructs final forest of tree–like structures. We used Flex version 2.5.35 and Bison version 2.4.3.

Outputs of those are combined into the PDDL Parser, so we place them into "PDDL_Parse" block as "flex" and "bison" modules. Flex needs an additional "FlexLexer" module for adaptation from C to C++ object model. It is provided by third party, so we can't change its namespace. Therefore it is presented on scheme (Pic 3.2) differently to keep consistency.

Finally we need a handler for calling the Parser, storing the parsed forest of tree–like structures, storing error collection and handling requests on the parsed forest. This handling is implemented within the "driver" module.

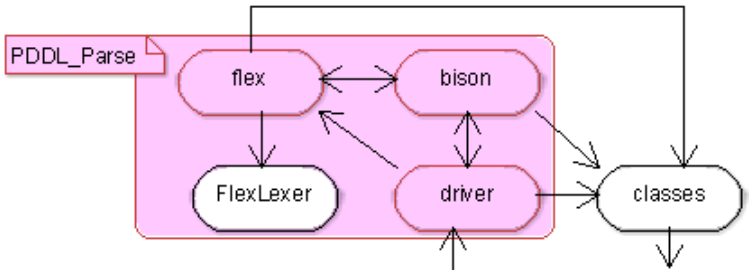


Pic 3.2 – Content of PDDL_Parse block, modules: "flex", "bison", "driver" and third party module: "FlexLexer"

Schema of cooperation between modules can be seen on Picture (Pic 3.3). Module "flex" utilizes: a) "FlexLexer" module to work with C++, b) "bison" module for tokens definition and c) "classes" module as some of tokens are complete elements of the PDDL (therefore class definitions in this module). Module "bison" utilizes a) "classes" module for nodes from which it constructs forest of tree–like structures, b) "flex" module as source of tokens and c) "driver" module for entire forest storage and error collection for Error Detection feature. Module "driver" provides "flex" module with PDDL code to be parsed and initiates "bison" module for parsing. It uses definitions from "classes" module.

"PDDL_Parse" workflow is as follows:

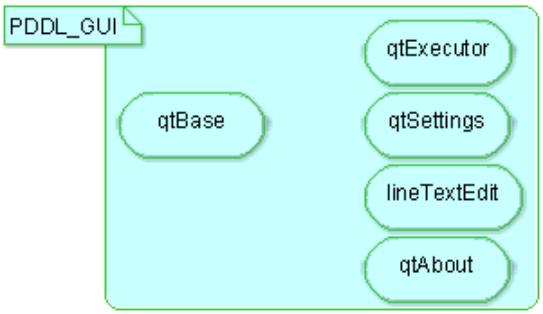
- When an external module needs to run the Parser, "driver" module makes initializations and runs the parsing process, "flex" module takes PDDL code at its input and produces stream of tokens, "bison" module takes stream of tokens at its input and produces forest of tree–like structures. Each tree–like structure "bison" module completes is registered into forest in "driver" module. Similarly when "bison" module encounters an error that cannot be placed into tree–like structure, it reports it into collection in "driver" module.
- When an external module needs to utilize some Parser dependent feature, "driver" module distributes the call onto the parsed forest and returns the result.



Pic 3.3 – Cooperation of modules in "PDDL_Parse" block, arrow points to module which is being utilized

3.5 Block "PDDL_GUI"

To provide a quality GUI we decided to use a third party framework. Possible candidates were GTK [24], Qt [19] and .NET [1]. We found Qt community to be more alive with faster development and better accessible documentation [16], so we decided for this framework. Qt version we used is 4.7.2. The "PDDL_GUI" block consists of these modules: a) "qtBase", b) "lineTextEdit", c) "qtExecutor", d) "qtSettings" and e) "qtAbout". All modules from "PDDL_GUI" block can be seen in picture (Pic 3.4).



Pic 3.4 – Content of "PDDL_GUI" block, modules: "qtBase", "qtExecutor", "qtSettings", "lineTextEdit" and "qtAbout"

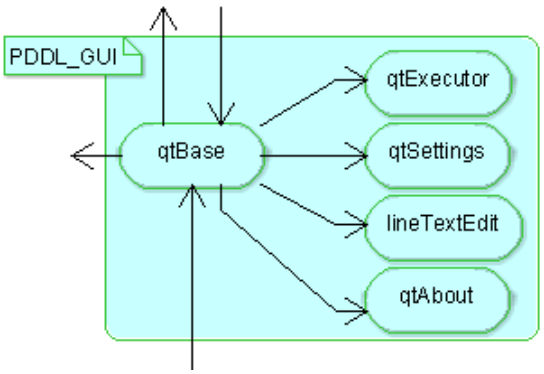
Major module of "PDDL_GUI" block is "qtBase" module as it presents main window of the application. Project Management feature is implemented in this module as well as interconnection of features with GUI. Another things implemented in this module are Interactive Error Table and Auto–Save feature. This module contains all application's timers:

- Parser timer – a typical developer quickly writes few key–strokes and then pauses typing for some time. During the writing period he/she shouldn't be burden by the Parser whereas during the pause he/she may want to check the result of Parser dependent features. Therefore we need a timer activated Parser, where any key stroke restarts the timer. Timer interval should be configured to reset before timing out when a developer writes, yet time out quickly during a pause period to provide a developer with the actual data. The interval should be modifiable by a developer, the default one is 500 ms which we experimentally found as satisfactory.
- Auto–Save timer – a time after which backup files are updated, the interval should be modifiable by a developer, the default one is 1 min.
- Error check timer – a time after which all project's files are checked for errors, the interval should be modifiable by a developer, the default one is 20 s.

Module "lineTextEdit" serves as major editing element of our tool, it is the editing window. Code Completion pop–up presentation is implemented here as well as Code Collapsing feature with Line Counter. Presentation of highlights from Syntax Highlighting, Error Detection and Bracket Matching features are processed by framework in this module.

Third party software integration resides in "qtExecutor" module and ability to change highlighting colors resides in "qtSettings" module. These modules also contain graphical window for its features. Every program needs some "about" window. We implemented this in the "qtAbout" module.

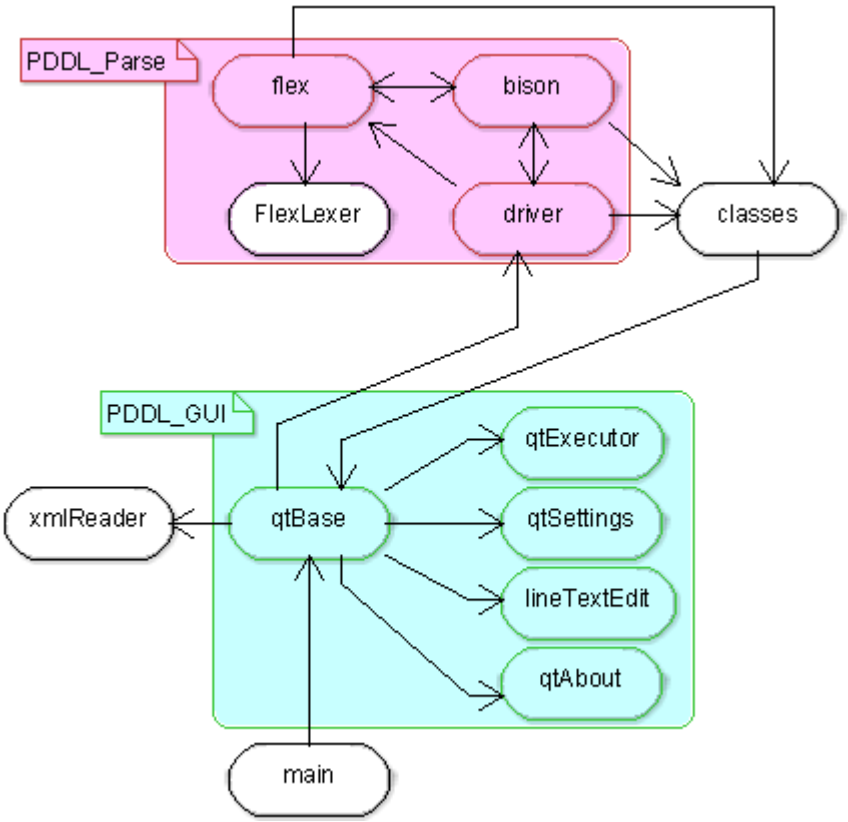
Picture (Pic 3.5) shows cooperation between modules. Module "qtBase" handles all others modules in this block. It operates tightly with "lineTextEdit" module (e.g. sets text, gets text, set is highlighting), as it is most important visible element of this module. When developer needs to a) utilize integration of third party software, control is passed to "qtExecutor" module, b) change settings, control is passes to "qtSettings" module and c) see about window, control is passed to "qtAbout" module. "qtBase" module interconnects whole "PDDL_GUI" blocks with the rest of the application.



Pic 3.5 – Cooperation of modules in "PDDL_GUI" block, arrow points to module which is being utilized

3.6 Interconnection of application modules

Module "main" is started as the first module of the application. After the initialization it passes control to "qtBase" module. Module "qtBase" utilizes "xmlReader" module to import XML if requested and "driver" module to interconnect blocks "PDDL_Parse" and "PDDL_GUI". Module "classes" utilizes module "qtBase" for passing graphical information from computational part of GUI based features. Complete scheme can be seen on picture (Pic 3.6).

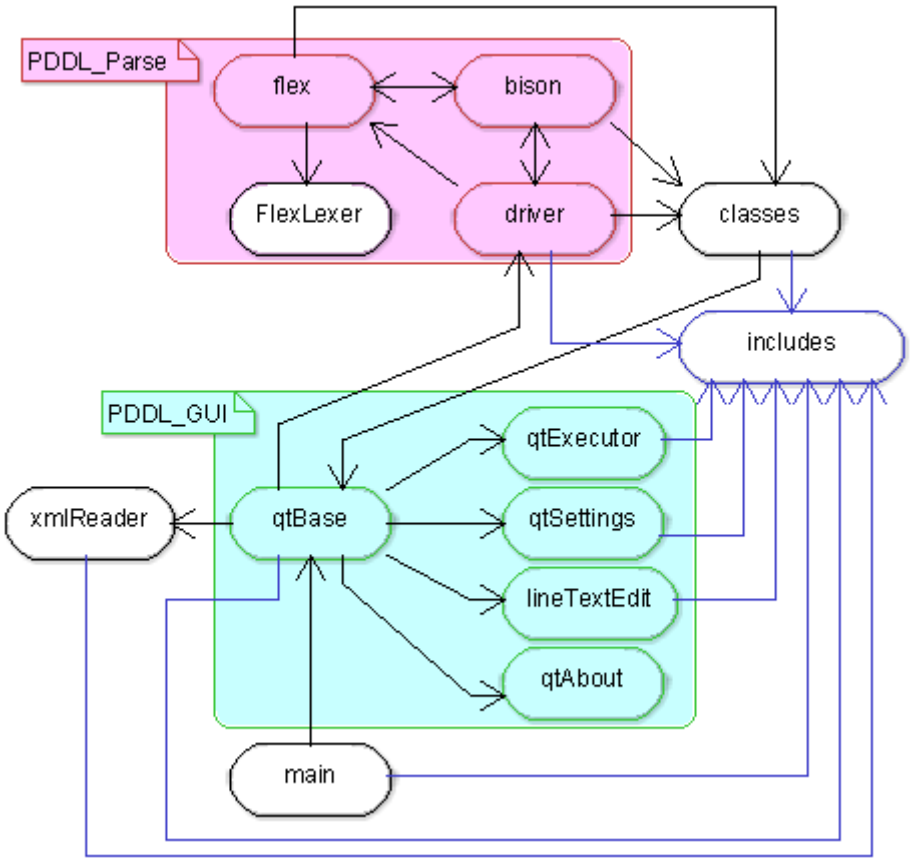


Pic 3.6 – Interconnection of all modules, arrow points to module which is being utilized

3.7 Common libraries and Runtime Logging

Because many modules utilize some of standard libraries, we decided for centralized inclusion of them for better control. All standard libraries includes are present in "`includes`" module made for this purpose. This module is utilized by each other module except "`qtAbout`", "`flex`", "`bison`" and "`FlexLexer`" modules.

Considering "`includes`" module utilized by almost each other module, we implemented logging feature here. This way we have single point of logging macro operation, so during compilation without logging feature, whole logging definition can be omitted. Final structure of our architecture can be seen on picture (Pic 3.7).



Pic 3.7 – Structure of our tool's architecture, arrow points to module which is being utilized

4 Implementation

In this chapter we inspect our application module by module with focus on implementation details. To provide the future developer of our tool with even more options, our code is well commented and compatible with automated source code documentation generating tool, Doxygen version 1.7.4 [7]. In this chapter we proceed in the following order:

- Module "main"
- Module "qtBase"
- Module "lineTextEdit"
- Module "qtSettings"
- Module "qtExecutor"
- Module "qtAbout"
- Module "xmlReader"
- Module "driver"
- Module "bison"
- Module "flex"
- Module "FlexLexer"
- Module "classes"
- Module "includes"

4.1 Module "main"

Module "main" is the first module executed on the program start. This module consists of sole definition file "main.cpp". Two primary roles of this module are handling of the configuration file and initializing the Qt environment (initializing "qtApplication" class) before passing execution to "qtBase" module from "PDDL_GUI" block.

Handling of the configuration file takes place in "loadConfig()" method. If the file "pddl.config" exists and can be opened, configuration is loaded from it. Otherwise handling method creates default one and sets default configuration. Configurable values are:

- Parser time – idle time after which the Parser is started
- Auto–Save time – how often to backup modified files
- Error check time – how often to check all files for errors

4.2 Module "qtBase"

Module "qtBase" is major part of "PDDL_GUI" block. This module manages main window of application and interconnects it with non–graphical parts of the program. This module consists of a header file "qtBase.h", a definition file "qtBase.cpp", a GUI elements file "qtBase.ui" and generated C++ files from the GUI file. Responsibilities of this module are:

- Initialization – initializes program environment
- Interconnection with source code editor – source editor has its own module "lineTextEdit"
- Handling of other GUI modules – except "lineTextEdit" each module from "PDDL_GUI" block presents one application window

- Parser and coupled features – because many features are parser dependent, they are handled together with parsing
- Project Management – project management is implemented in this module
- Handling of other features – other features implemented in this module and features initiated from this module
- Timers – some features are initiated by timer tick, their timers are located in this module

4.2.1 Initialization

Initialization is the first action of this module. Initialization of graphical elements is handled by generated files from GUI elements file ("`qtBase.ui`"), rest of the initialization is implemented in the "`qtBase`" class constructor, the only class of this module.

After initialization of graphic elements, those are connected onto appropriate methods, except the menu. Menu elements are connected to method "`menuAct()`" which handles them all. Timers are set according to the setting from configuration file, processed in "`main`" module. All child members are initialized and finally startup no–project state is achieved.

4.2.2 Interconnection with source code editor

The source code editor element is instantiation of "`lineTextEdit`" class from "`lineTextEdit`" module. Interconnection consists of these tasks:

- Acquire text content when needed (e.g. Parser run).
- Change text content when needed (e.g. change of the active file).
- Acquire cursor position (e.g. for status bar, for Code Completion).
- Detection of cursor relocation for status bar update and Bracket Matching. Detection of changes, so timers can be triggered and file marked as modified. To achieve these we registered signals of "`lineTextEdit`" module onto slots of this module: "`cursorPositionChanged()`" signal to "`textPosChanged()`" slot and "`textChanged()`" signal to "`textChanged()`" slot.
- Update information for parts of features handled in the source code editor (e.g. Syntax Highlighting visualization, Code Collapsing borders, Code Completion completion list).
- Font update when developer changes the font.

4.2.3 Handling of other GUI modules

Other modules from "`PDDL_GUI`" block (except "`lineTextEdit`") represent single application window and are instantiated as a child of this class during the initialization. Therefore when a developer uses control which summons other window, the execution is passed to it.

4.2.4 Parser and coupled features

Initiation of parsing process and coupled features resides in "`ProcessParsed()`" method. Most collections storage and Parser handling are implemented in "`driver`" module. During the whole process, the status bar informs a developer about the parsing process being active. Execution order is as follows:

- Parsing. After clearing of old forest by "`clear()`" method of "`driver`" module, parsing itself is initiated in "`ParseText()`" method. It extracts

text from the source code editor and passes it to "parse()" method of "driver" module. On return, new parse forest is created and parse time errors are collected in the error collection, however exact text position of some errors may be missing.

- Highlighting first phase. Highlighting is made by collecting highlights into the collection in few phases and then passing collection to the source code editor. First phase is clearing the highlight collection and calling Bracket Matching routine (without applying).
- Error Detection second phase. Parsed forest is checked for errors. To determine also semantic errors, parsed forests of other project's files can be searched for additional information (e.g. requires statement of domain, defined predicates in domain).
- Highlighting second phase. Second phase calls "Process()" method of "driver" module, which collects major highlights from the forest and collected errors. This phase utilizes "colorer()" method for highlights collection. Additionally during run of this method, "updPos()" method can be called for completing a position information in collected errors. Finally the highlights collection is passed to the source code editor.
- Placement of collected errors into Interactive Error Table is implemented in "showErrors()" method.
- Rebuilding of completer list for Code Completion feature. Cursor position is passed to "getCompleterType()" method of "driver" module which returns type of completion list. If the type of completion list is collection of developer defined words, additional call to "driver" module is made, to collect appropriate words. Completer list is then passed to the source code editor.
- Code Collapsing update. Code Collapsing data pointer is retrieved from the source code editor and passed to "driver" module for matching with new parsed forest. To refresh collapsed state "blockmaster()" method on the source code editor is called.

4.2.5 Project Management

Project Management handles various tasks. Some of them work with project's files (e.g. removing a file), so these firstly check whether a valid project is opened and an active file is selected. If not, they show error and return from method. These checks are not included in following description. Also modified status of project's files is stored as actual file's content, so unmodified files have this variable set to null. Reading modified file reads its variable while reading unmodified file loads it from the persistent storage.

Tasks handled by Project Management are:

- Selecting the active file. It resides in "listFileSelect()" method and utilizes list of project's files, which works also as a switch for file selection. It ensures that old active file's content is stored in a variable. Then it loads content of the new one into source code editor and selects context in "driver" module and source code editor by their "selectCurrent()" methods. After the switch, "ProcessParsed()" method is called to parse active file and execute Parser dependent features. During whole process, flag for ignoring changes is set to prevent routines registering text updates to proceed.

- Synchronizing list with project's files. This task resides in "`syncList()`" method. It synchronizes visible list of project's files with actual project's files. Each record in list also contains whether the file is modified and count of detected errors in the file. Synchronization works by erasing previous data of list and searching all project's files. For each project's file, context of module "`driver`" is swapped and checked for error count and modified status. At the end context of module "`driver`" is reset to active file.
- Preventing accidental data loose. "`preClear()`" method indicates whether risk of losing data is present or not. Situation is considered unsafe when collection of project's files was changed or any of project's files is modified. If situation is considered unsafe, developer is allowed to make decision whether ignore it or interrupt action. Method returns developer's selection on unsafe state and "`true`" on safe state. Any method utilizing this should abort its execution when "`false`" is returned.
- Cleaning process. This process resides in "`doClean()`" method and it ensures that no rubbish data from previous project remains in memory. It locks and clears source code editor and call "`ClearAll()`" method in "`driver`" module. Then it clears highlights collection, stops Parser timer, removes all temporary backup files and erases modified status for all project's files. It clears collection of project's files, sets previous completer type to null and synchronizes list of project's files. Finally it clears Interactive Error Table.
- Creating a new project. This task resides in "`newProj()`" method. It prevents accidental data loose, executes the cleaning process, creates a new project file, changes working directory and synchronizes list with project's files (into empty state).
- Opening the project, implemented in "`openProj()`" method. At first the process is similar to creating a new project. It prevents accidental data loose, executes cleaning process, opens selected project file and changes working directory. Then it reads the project file for project's files. For each project's file, the existence of backup file from Auto–Save feature is checked and if exists, its content is loaded as actual content of the file, which is also set as modified. After loading of all files, "`allErrCheck()`" method is called to check for errors, which also synchronizes list with project's files.
- Closing the project. It is implemented in "`closeProj()`" method, prevents accidental data loose and executes cleaning process.
- Adding new file into project. It is implemented in "`newFileToProj()`" method. It creates developer's wished file, uses "`tryMakeRelative()`" method to acquire relative path to the file and add it into project's files. At the end it synchronizes list with project's files.
- Adding existing file into project, which resides in "`existingFileToProj()`" method. It uses "`tryMakeRelative()`" method to acquire relative path to the file and adds it into project's files. Context in "`driver`" module is switched by "`selectCurrent()`" method to newly added file and file content is parsed. At the end, list with project's files is synchronized.
- Excluding a file from a project, which resides in "`removeFileFromProj()`" method. Source code editor is set to blank

and locked, file being an active file is erased from project's files, active file is selected to none and list with project's files is synchronized.

- Saving an active file, residing in "`saveFile()`" method. If file is marked as changed, its content is saved and temporarily backup file is removed. List with project's files is synchronized, as it presents also modified status of files.
- Saving a whole project resides in "`saveProj()`" method. If project's files were changed, the project file is saved. For each project's file, if it is marked as changed, its content is saved onto disk and corresponding backup file is removed. List with project's files is synchronized, as it presents also modified status of files.

4.2.6 Handling of other features

As this module resembles the main window of the application and contains Project Management, many other features are implemented and some initiated from this module. Features handled from this module are:

- Error check on all files. It resides in "`allErrCheck()`" method. It swaps context of "`driver`" module for each file, clears error collection, initiates parser and calls second phase of Error Detection. At the end, it resets the "`driver`" module context and calls method "`syncList()`" for synchronization of list with source code files. A developer is informed about the process being executed in the status bar.
- Interactive Error Table. It shows all found errors: location of the error detection, which is the most probable error location, the originator of the error and brief description of the error. Developer is allowed to alphabetically sort errors by any of the fields in the record, and double click any of them, so cursor position in source code editor jumps into the location of the error detection. Filling error table resides in "`showErrors()`" method and reacting to developer's request to skip onto error resides in "`errorDblCick()`" method.
- Bracket Matching. Bracket Matching resides in "`bracketMatcher()`" method. It retrieves text position from source code editor, utilizes "`bracketPartner()`" method of "`driver`" module, updates highlights collection and can apply it.
- XML export. XML export feature resides in "`outXML()`" method. After a developer selects an output file, it calls "`outXML()`" method of "`driver`" module and stores its output along with header containing the source file name into the selected output file.
- XML import. This feature resides in "`inXML()`" method. It lets a developer to select an input file and then instantiates "`xmlReader`" module with content of that file. "`xmlReader`" module then creates PDDL output which is inserted into the source code editor.
- Auto–Save feature, implemented in "`safetySave()`" method. It simply saves the content of all modified and unsaved files into backup files with extension "`pddl_bckp`". During saving process, the status bar informs developer about the operation.
- Status bar information. Actual position information are set into the status bar by "`posShowUpd()`" method. Some locking features also update the status bar to inform developer about the reason for an application lock.

- Font changing feature is implemented in "changeFont()" method and works by calling a font select dialog. Font selection is then applied to the source code editor.

4.2.7 Timers

Some features are initiated by timer tick, so these need a timer. Three timers are located in this module:

- Parser timer, which starts parsing process. This timer is reset each time a change in the source code editor is detected.
- Auto–Save timer, which saves unsaved files into backup files.
- Error check timer, which initiates error check on all files.

4.3 Module "lineTextEdit"

Module "lineTextEdit" represents the source code editor element. This module consists of a header file "lineTextEdit.h" and a definition file "lineTextEdit.cpp". It is part of "PDDL_GUI" block and contains two classes: "lineTextEdit" main class derived from "QPlainTextEdit" Qt's base class and "lineCounter" helper class derived from "QWidget" Qt's base class. The helper class supports the Line Counter feature. Main class's role derived from the base class is text editing including all the standard features like copy, paste, undo, redo and text highlighting. We added additional roles for our purposes:

- Line Counter
- Code Completion
- Code Collapsing

4.3.1 Initialization

Initialization of main class consists of following:

- Instantiation and initialization of "QCompleter" child class for Code Completion
- Width determination of number 9 for Line Counter as we found no other number to take more place than this
- Instantiation and initialization of "lineCounter" child class, which is the actual counter
- Connection of signals to appropriate slots.

4.3.2 Line Counter

Line Counter utilizes the helper class, which represents the counter itself. It is then painted into the main class which has its text moved to right by "spaceSeize()" method so it does not collide with the counter. For this purpose the helper class reports its "sizeHint" value by main class's "spaceNeeded()" method which computes the needed space from number of digits needed to represent the last line.

Printing counter numbers into the helper class is implemented in "printCntr()" method, which creates "QPainter" object and searches through all blocks located above the bottom end of the source code editor, where one block represents one line of a text. If block is visible, its "y" coordinate is retrieved and corresponding line number is painted on the correct position. Also opening and closing switches for Code Collapsing is painted this way.

Because the helper class is tightly coupled with the main class, "resizeEvent()" slot of main class is modified not only to call

"resizeEvet()"base slot, but also to set a geometry of the helper class. On repaint "upateLineCounter()" method ensures correct painting of the line counter by calling its "update()"or "scroll()" method depending on whether text was scrolled or not.

4.3.3 Code Completion

Completing a word is handled by "insertCompletion()" method. This method first extracts a text cursor, finds start and end of the line, determines borders of current word prefix, corrects end of the word prefix with use of "charValid()" method, moves the cursor at the correct end of the word prefix, inserts a completion and moves the visible cursor to the end of the word. Because PDDL language uses some special characters we made "charValid()" method to recognize valid characters of a word.

Change of the Code Completion list is handled by "setComplModel()" method. After the list is changed, the method extracts a current text prefix with "getPrefix()" method and if it differs from the previous one, it updates it and set index in the completer window to first possibility. Also if the completer popup is hidden, this method makes it to show.

We want certain keys to be evaluated within the completer popup and not in the source code editor major text area. Evaluating corresponding keys and preventing their spread to the source code editor text area is handled in "keyPressEvent()" slot. It checks whether current key is one of those that are fully handled by the completer, if yes, the key event is ignored from this moment and the method returns, otherwise the key event is forwarded to the base class. Next it checks whether the key event represents a shift key only, and returns if yes. Then it extracts prefix from the text. If the prefix is empty or escape or control key is pressed, the popup is hidden and the method returns. If the prefix differs from the previous one, it is updated in the completer class and the completer popup is shown.

4.3.4 Code Collapsing

Maintaining of blocks visibility is implemented in "blockMaster()" method. This method starts by acquiring the first visible block and creating a variable for dept with initial value of zero. Then it searches all valid blocks, where one block represents one line of a text. For each block it sets visibility based on the depth value and checks collapsing data structure whether any depth change is needed. When the collapsing structure presents an active collapsing region start, the depth is increased, when the depth is nonzero and the collapsing structure presents an inactive region start, the depth is increased too, because we need to count with the end region tag. The depth is decreased when it is bigger than zero and the collapsing structure presents the end region tag. Last block of the text is always visible, as program tends to crash when it is hidden. Finally a fake resize event is created, because "QPlainTextEdit" base class is not fully prepared for any collapsing and some cached values will not be recomputed which inflicts bad behavior of a scroll bar and can cause text inaccessibility.

Changing of a folded status is managed by "foldChange()" method. This method finds the correct block by vertical position and swaps active to inactive tags in the collapsing data structure which belongs to this block, and vice versa.

Code Collapsing switches are printed in "printCntr()" method along with Line Counter numbers. Catching interaction of collapsing switches is handled from

a helper class in "mousePressEvent()" slot, which calls "foldChange()" method of the main class with appropriate vertical position.

For switching collapsing data structure context "selectCurrent()" method is used, so collapsing of regions is persistent between project's files. For passing of the collapsing data structure via "qtBase" module to "driver" module for update "retCollapsibles()" method is used. For consistency between the source code editor and "driver" module prior to an update, whenever line count changes, "adjustByBlockChange()" slot compares current lines count with previous and moves or erases influenced regions accordingly.

4.4 Module "qtSettings"

Module "qtSettings" is used for changing highlights colors and is part of "PDDL_GUI" block. This module consists of a header file "qtSettings.h", a definition file "qtSettings.cpp", a GUI elements file "qtSettings.ui" and generated C++ files from the GUI file. During its construction it obtains a pointer to actual formats for highlighting, so it can easily change them, and connects buttons signals to appropriate slots.

Each highlight type change is started by its own slot which calls "changeColor()" method. It presents a developer with color select dialog and if valid color is selected, corresponding highlight style is changed. Finally "refreshPreview()" method is called to ensure that all preview texts are showed with correct highlight.

4.5 Module "qtExecutor"

Module "qtExecutor" handles an integration of third party software. It is part of "PDDL_GUI" block. This module consists of a header file "qtExecutor.h", a definition file "qtExecutor.cpp", a GUI elements file "qtExecutor.ui" and generated C++ files from the GUI file.

Upon initialization this module connects all button signals to appropriate slots and stores a read only pointer to project's files so it can access their names and relative paths. Save and load features resides in "saveList()" and "loadList()" methods. Executions of commands take place in "executeList()" method, which starts by obtaining commands as a text, replacing patterns presenting file names by their names and then by replacing patterns presenting directory path by the actual working directory path. Finally the text is processed line by line, for each line appropriate command is sent to operating system.

4.6 Module "qtAbout"

Module "qtAbout" is part of "PDDL_GUI" block and consists of a header file "qtAbout.h", a definition file "qtAbout.cpp", a GUI elements file "qtAbout.ui" and generated C++ files from the GUI file. This module handles appearance of "about" window.

4.7 Module "xmlReader"

Module "xmlReader" handles XML import feature. It consists of a header file "xmlReader.h" and a definition file "xmlReader.cpp". On initialization it

stores a source stream which is later used, so it is one time use class. Main method is "read()", which works by repetitive calling "getPart()" method, output value from which is directly translated to PDDL. Method "getPart()" obtains one token from the source and returns it, for this purpose it utilizes "blank()" method which decides whether character is part of a word or not.

4.8 Module "driver"

Module "driver" mainly interconnects "PDDL_GUI" block containing graphic and management parts of program with "PDDL_Parse" block which handles parsing process and "classes" module which represents individual language parts. This module consists of a header file "driver.h" and a definition file "driver.cpp".

Parsed forest of tree–like structures and collected errors are stored in this module, "qtBase" module retrieves errors by "PassErrors()" method. One parsed forest is stored for each context, which represents one project's file. Parsed forest is divided into three parts: "problems", "initsits" and "domains". Some tasks are forwarded here from "qtBase" module for their distribution to the parsed forest.

Tasks of this module are:

- Run parser
- Select current context
- Collect initial situations and domain names
- Error Detection
- Determine completer list
- Process highlighting
- Find matching bracket
- Register collapsing regions
- Generate XML output
- Clear current context
- Clear all data

4.8.1 Run parser

When "qtBase" module asks for parsing by calling "parse()" method, it initializes "flex" module, provides it with a data and the Parser is run by "parse()" method on "bison" module. During run, "bison" module utilizes tokens provided by "flex" module and on demand registers new trees or errors into the parsed forest by "register_new()"overloaded method. Trees are simply placed into correct part of the forest while new errors are firstly checked whether they are non–zero pointers.

4.8.2 Select current context

As this module stores a parsed forest and an error collection for each context, it has to be changed with an active file change. It is achieved by "selectCurent()" method, which redirects references "domains", "problems", "initsits" and "errors" to corresponding positions in collections "domainsALL", "problemsALL", "initsitsALL" and "errorsALL".

4.8.3 Collect initial situations and domain names

Some features utilize this information, so "collectDomainNames()" and "collectInitSitNames()" methods searches appropriate part of the parsed

forest and collects desired information by "CollectName()" method of "classes" module.

4.8.4 Error Detection

When "qtBase" module asks for additional Error Detection, "ErrCheck()" method is called. This method searches through the parsed forest and calls "ErrCheck()" method on each tree root. Prior to calling "ErrCheck()" method on "problems" part of the forest, its corresponding domain is retrieved by "giveDomainName()" method. Each tree root represents some class from "classes" module. Found errors are collected by "register_new()" overloaded method.

4.8.5 Determine completer list

Firstly "getCompleterType()" method initializes result to null. Then it calls "getCompleterType()" method on each tree root on the parsed forest. Only tree which hits is expected to change the result. Prior to processing "problems" part of the forest, its corresponding domain is retrieved for additional parameters.

4.8.6 Process highlighting

When "qtBase" module asks for highlights collection, "Process()" method searches through the full parsed forest and calls "Process()" method on each tree root as well as on each collected error. During run, highlights are collected by calling "colorer()" method in "qtBase" module.

4.8.7 Find matching bracket

When "qtBase" module asks for a matching bracket, it does so by calling "matchBracket()" method. The method initializes result to known bracket and searches through the full parsed forest calling "matchBracket()" method on each tree root. Only tree which hits is expected to change the result value, others should not change it.

4.8.8 Register collapsing regions

Collapsing regions are context aware, therefore it is needed to match them with the actual parsed forest and this is made via "UpdateCollapsibles()" method. This method starts by collecting current collapsing regions from the parsed forest by "Collapsibles()" method, and then it compares current regions with old regions passed in a parameter as reference. When match is found, the old state of the active/inactive collapsing region is preserved. Then current collapsing regions are copied into "lineTextEdit" module by the same reference.

4.8.9 Generate XML output

When "qtBase" module asks for XML output, "outXML()" method is called. This method searches through the full parsed forest and calls "outXML()" method on each tree root. As it searches through the full parsed forest part by part, a corresponding XML tag is added before and after each part.

4.8.10 Clear current context

Before proceeding with data collecting, it is wise to dispose old data. This process takes place in "Clear()" method. Clearing method clears collected errors and all three parts of the forest of the current context.

4.8.11 Clear all data

At certain situations it is needed to refresh "driver" state into startup status, this can be achieved by calling "ClearAll()" method. This method clears all collected errors and all parts of parsed forests in all contexts. It clears all contexts as well.

4.9 Module "bison"

Module "bison" is the main parser module and is part of "PDDL_Parse" block. This module consists of a generated header file "bison.h", a generated definition file "bison.cpp" and a grammar file "bison.y".

This module takes tokens from "flex" module and transfers them into classes from "classes" module depending on a grammar in the grammar file. Most created classes are just children for some other classes, only few major classes are considered final trees for the parsed forest. Child classes are registered to their parent classes by calling corresponding method on the parent, during this process some errors could arise. Errors that could arise during the child to parent registration are handled by the return value, where no errors are represented by the zero pointers, so return value is passed to "register_new()" overloaded method in "driver" module. Each created final tree is registered by "register_new()" overloaded method in "driver" module.

Because this Parser works in interactive program not as a batch program, it is important to recover from possible errors and continue in the parsing of the rest, so the grammar must be adapted to this fact. Detected parse time errors are collected by the "register_new()" overloaded method in "driver" module. Recognized partly unfinished structures are those, which: a) prematurely ends by end of input, b) are missing some substructure entirely or c) contains partly unfinished substructures. Each present left bracket must be present with its matching right bracket (excluding right brackets near the end of the input) for Parser to work correctly.

4.9.1 Grammar file

Grammar file is used for generation of the header and the definition file. This file is expected to be modified during a) an update of known PDDL version or b) adding other features. Rule of the thumb: One major type in the grammar roughly equals to one class in "classes" module. Note to know, PDDL as it is defined has some ambiguities while we need our grammar to be unambiguous and deterministic. Therefore few clarifications are done. Also when editing the grammar, consider all possible predictable errors and include them into the grammar to avoid unrecoverable situations.

Grammar is strongly dependent on the correct bracketing, other errors are mostly recoverable. For this purpose we use "whatever" token, which matches with any correctly bracketed expression. Tokens are prefixed with "T_" and types are prefixed with "nt_" for normal type and "wt_" for equivalent type allowing being omitted. Type allowing being omitted is precaution to achieve recoverable grammar, as well as normal types with multiple definitions: correct and with predictable errors.

Grammar file starts with a verbatim code part, where inclusion of "classes" module is. After the verbatim code part are some settings for generating files: file names, a switch for better error texts, namespace definition, class name, "driver" interface and start symbol. Next is a union of used types, which are classes from

"classes" module and "integer". After the union are tokens which correspond to "flex" module and types we use in our grammar, which are based on union's classes. Types are followed by destructors for them and another verbatim pasted code. The inclusion of "flex" and "bison" header files as well as connection of "flex" module to our "bison" parser. Final part of the grammar file is the grammar itself followed by the error handling method. Error handling method calls "register_new()" overloaded method of "driver" module.

4.10 Module "flex"

Module "flex" which is part of "PDDL_Parse" block works as a preprocessor for "bison" module. It takes PDDL and processes it into tokens which are passed to "bison" module for parsing. This module consists of a header file "flex.h", generated definition file "flex.cpp" and Flex's source file "flex.l".

Flex source file starts with a verbatim pasted code which consists of a header file inclusion, inclusion of tokens from "bison" module and definition of "termination" token. After the verbatim code part are some settings for file generation: switch for C++ generation, class name, file name, switch for Windows usability, switch for parsing of included files and switch for case sensitivity. Finally, there are regular expressions for tokens in the Flex source file followed by a second verbatim code part. This verbatim code part contains: a) constructor, b) destructor, c) macro redefinition and d) two methods.

In this implementation, "flex" module returns each keyword as a token, few special characters as a token and brackets as a token. Comments are not returned in any way, so their processing ends here. Expressions which fall all these, are returned as number, variable, text or by colon prefixed text token.

4.11 Module "FlexLexer"

Module "FlexLexer" is helper module for "flex" module to work with "C++" object model, as tool flex is written for plain "C" language. This module is standard module obtained from Flex's files. This module contains sole header file "FlexLexer.h".

4.12 Module "classes"

Module "classes" can be considered the heart of the PDDL Studio. It consists of a header file "classes.h" and a definition file "classes.cpp". This module contains:

- "posType" structure for position storage
- "abstrGui" abstract base class for a GUI (in this case "qtBase" module)
- "ErrorType" structure for error storage
- "ParseTmpStruct" structure used as help structure in parser
- "params" structure utilized during Error Detection
- one class per each element of PDDL language

4.12.1 Structure "posType"

Structure "posType" us used for storing an information of word's or any element's position in a text, it contains starting position and ending position. Both positions store information about line, column and character from the start of the file. It has

implemented "move()" method for movement by certain number of characters and "newLine()" method for new line and movement by certain number of characters. Both movement methods utilize "setCurAsPrew()" private method to store the end position as the start position before a movement.

4.12.2 Abstract base class "abstrGui"

Because a class derived from this class is expected to have a method for passing highlights, these are implemented here in a collection along with "colorer()" virtual method. "updPos()" virtual method is prepared here, for updating of a position information. Also determination of a completer list type is utilized by a class derived from this class, however it takes place in this module, so types of completer lists are implemented here completely.

4.12.3 Structure "ErrorType"

Structure "ErrorType" serves as a single error storage, so it stores an error position, a text describing error and an error originator. Also this structure contains "Process()" method which updates the position information if it is incomplete and registers highlight for the error, both by calling appropriate methods in "qtBase" module.

4.12.4 Structure "ParseTmpStructure"

Structure "ParseTmpStructure" is simple structure containing four integer numbers and four void pointers and is used during parsing as a helper storage structure.

4.12.5 Structure "params"

Structure "params" is utilized mostly during Error Detection and Code Completion list determination, where it stores an additional information needed for correct evaluation. The additional information consists of requirements flags defined in a domain definition, defined constants, types and predicates. It contains also action parameters and action variables.

4.12.6 Classes for PDDL elements

Each element in PDDL language has its corresponding class here. Sometimes more derivates of the base class depending on the element variability are present. These are used by "bison" module for forest construction, so usually update of this module requires update of "bison" module as well. Every element consisting of other elements is implemented by class which contains child variables of corresponding types.

Few methods are expected to be implemented in each class. Each of these methods makes computation within its node and calls corresponding methods on its child nodes. These methods are:

- "ErrCheck()" method which checks for errors in the current element and collects them by "register_new()" method in "driver" module. When Error Detection is dependent on some parameters from other elements, the class representing root of the tree in which these element could be, contains "SetParams()" method, which sets "params" structure according to the current element.
- "Process()" method which registers its highlights via "colorer()" method of "qtBase" module.

- "outXML()" method, which handles making of the XML output. Some classes contain "lineXML" method, which is similar to previous method, but omit formatting, so its parent can do formatting itself.
- "Collapsibles()" method, which registers its intended collapsing regions into a structure passed as a parameter and utilizes a text cursor, which is passed as a parameter too, for converting an absolute position (which is known to the node) to the line number (which is not).
- "matchBracket" method. This method is expected to first check whether first bracket is owned by current object and set corresponding partner bracket or proceed without altering output variable.
- "getCompleterType()" method, which first checks whether cursor is within an element it is representing or not. If yes, it determines the best fitting completer list type. When completer list determination is dependent on some parameters, it is handled similarly to Error Detection.
- "startPos()" and "endStartPos()" method for determining position of detected error in that particular class
- "collect()" or "collectName()" methods which collect name of represented element. One of these methods is presented in these nodes that represent elements expected to be collected. Collection is made for setting parameters for Error Detection or determination of completer list type for Code Completion. Collection is made also for collecting completion list itself.
- Other methods that are used mostly during construction of parsed forest or are unique for a certain class because of nontrivial element it represents.

4.13 Module "includes"

Module "includes" encapsulates most external included libraries and defines clauses with global effect, so they are not spread through the all modules. This module consists of a header file "includes.h" and a definition file "includes.cpp". Moreover this module contains a debugging structure. Whether debugging is used or not is determined by a switch in define clauses with global effect in this module.

4.13.1 Debugging structure

Debugging structure is a simple class, which stores a static nested level. On its construction it reports and stores name from a parameter. It also and increases the static nested level. Analogically on its destruction it decreases the static nested level and reports its stored name. Reports are made to "std::cerr" and the nested level is used to prefix report with spaces.

Debugging is handled by a simple macro, which makes a local variable with debugging class, which gets a current method name as a parameter therefore each entry and exit from each method is logged.

Conclusion and Future Work

The goal of this thesis is the creation of an application capable of inspecting, analyzing and modifying PDDL documents. We successfully accomplished thesis goals by creating an application, which: a) supports Syntax Highlighting, b) does syntax and semantic Error Detections, c) has an integrated Project Management, d) is capable of XML export and import from its own XML format, e) supports Code Collapsing, f) supports Code Completion with contextual hints and g) contains a PDDL Parser, which can be used as a standalone Parser.

Our application is written in "C++" [6] and utilizes "Qt" [19] framework for GUI. PDDL Parser is made with help of third party tools: "Flex" [9] for lexical analysis and "Bison" [4] for parsing. Lexical analyzer from "Flex" produces tokens from its input and parser from "Bison" then uses these tokens for creation of parsed tree–like structure. Our application exceeds thesis goals in a few ways:

a) it supports third party software integration, so PDDL planners can be run from within the application,
b) it has implemented common editor features like Line Counter, Bracket Matching and Auto–Save feature for more convenient work,
c) it is multiplatform, thanks to used technology, therefore it influences wider range of developers,
d) it is standalone application, which, unlike plug–ins, is independent of, not always well documented, parent application,
e) it has modular architecture, so certain modules can be upgraded individually or entirely replaced in future development,
f) it implements Runtime Logging feature for easier future development,

We provided planning community with a tool designed for inspecting, analyzing and modifying PDDL documents, which is now ready to be deployed and to help development in the field of planning and artificial intelligence (AI). However we would like our tool to evolve beyond borders of this thesis. Main parts of our tool we would like to evolve are:

a) Parser – Parser implemented in current version of our tool is designed to respect PDDL version 1.2. Since this version, PDDL evolved into new version and acquired many new extensions (e.g. object–fluents) therefore the Parser should be updated to correspond to the actual state. Moreover because PDDL is expected to evolve in the future, the Parser should be maintained regularly.
b) Project Management – Project Management of our tool should be extended with new features, like file filters, ability to work with multiple projects at once and others to correspond with Project Managements of tools like Microsoft Visual Studio [25], Eclipse [8] or NetBeans [27].

Bibliography

[1] *.NET Framework – Wikipedia, the free encyclopedia.* [online]. Last updated 2012–04–04 [cit. 2012–04–05]. Avalable from WWW: <http://en.wikipedia.org/wiki/.NET_Framework>

[2] *Action description language – Wikipedia, the free encyclopedia.* [online]. Last updated 2012–04–18 [cit. 2012–04–18]. Available from WWW: <http://en.wikipedia.org/wiki/Action_description_language>

[3] Malik Ghallab, Dana Nau, Paolo Traverso. *Automated Planning: Theory and Practice.* Morgan Kaufmann Publishers. May 2004. ISBN 1-55860-856-7.

[4] Free Software Foundation, Inc. *Bison – GNU parser generator.* [online]. Last updated 2011–05–15 [cit. 2012–04–05]. Avalable from WWW: <http://www.gnu.org/software/bison/>

[5] Kautz Henry and Selman Bart. *BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving.* Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS–98. Pittsburgh, Pennsylvania. 1998.

[6] cplusplus.com. *cplusplus.com – The C++ Resources Network.* [online]. [cit. 2012–04–05]. Available from WWW: <http://www.cplusplus.com/>

[7] Dimitri van Heesch. *Doxygen*. [online]. Last updated 2012–02–26 [cit. 2012–04–05]. Avalable from WWW: <http://www.stack.nl/~dimitri/doxygen/>

[8] Eclipse Foundatin, inc. *Eclipse – The Eclipse Foundation open source community website.* [online]. Last updated 2012–04–07 [cit. 2012–04–07]. Available from WWW: <http://www.eclipse.org/>

[9] The Flex Project. *flex: The Fast Lexical Analyzer.* [online]. Last update 2008–02–26 [cit. 2012–04–05]. Avalable from WWW: <http://flex.sourceforge.net/>

[10] Vaquero, T. S., Silva, J. R., Ferreira, M., Tonidandel, F., Beck, J. C. *From Requirements and Analysis to PDDL in itSIMPLE3.0*. In Proceedings of the Third International Competition on Knowledge Engineering for Planning and Scheduling, ICAPS 2009. 2009. pp. 54–61.

[11] *ICAPS*. [online]. [cit. 2012–04–05]. Available from WWW: <http://www.icaps–conference.org/>

[12] *ICKEPS | ICAPS 2012*. [online]. Last updated 2012–04–18 [cit. 2012–04–17]. Available from WWW: <http://icaps12.poli.usp.br/icaps12/ickeps>

[13] *Imperative programming – Wikipedia, the free encyclopedia.* [online]. Last update 2012–04–11 [cit. 2012–04–14]. Avalable from WWW: <http://en.wikipedia.org/wiki/Imperative_programming>

[14] *Lisp (programming language) – Wikipedia, the free encyclopedia.* [online]. Last updated 2012–04–04 [cit. 2012–04–05]. Avalable from WWW: <http://en.wikipedia.org/wiki/Lisp_(programming_language)>

[15] Don Ho. *Notepad++ Home.* [online]. [cit. 2012–04–23]. Avalable from WWW: <http://notepad–plus–plus.org/>

[16] Nokia Corporation. *Online Reference Documentation.* [online]. [cit. 2012–04–05]. Avalable from WWW: <http://doc.qt.nokia.com/>

[17] D. McDermott et al. *PDDL – The Planning Domain Definition Language.* Tech Report CVC TR–98–003/DCS TR–1165. Version 1.2. Yale Center for Computational Vision and Control. 1998.

[18] Philippe Fournier–Viger. *PL–PLAN, an AI Planner.* [online]. Last updated 2009–07–06 [cit. 2012–04–07]. Avalable from WWW: <http://plplan.philippe-fournier-viger.com/>

[19] Nokia Corporation. *Qt — Qt – A cross–platform application and UI framework.* [online]. Last updated 2012–02–01 [cit. 2012–04–05]. Avalable from WWW: <http://qt.nokia.com/products/>

[20] Nau, D., Cao, Y., Lotem, A., Muñoz–Avila, H. 1999. *SHOP: Simple Hierarchical Ordered Planner.* In IJCAI–99, pp. 968–973.

[21] University of Maryland. *Simple Hierarchical Ordered Planner.* [online]. Last updated 2005–07–15 [cit. 2012–04–07]. Avalable from WWW: <http://www.cs.umd.edu/projects/shop/index.html>

[22] Simpson, R. M. *Structural Domain Definition using GIPO IV*. In Proceedings of the Second International Competition on Knowledge Engineering for Planning and Scheduling. 2007.

[23] *Syntax highlighting – Wikipedia, the free encyclopedia.* [online]. Last updated 2012–04–02 [cit. 2012–04–05]. Avalable from WWW: <http://en.wikipedia.org/wiki/Syntax_highlighting>

[24] The GTK+ Team. The *GTK+ Project.* [online]. [cit. 2012–04–05]. Avalable from WWW: <http://www.gtk.org/>

[25] Microsoft Corporation. *Visual Studio Home | Microsoft Visual Studio.* [online]. [cit. 2012–04–05]. Avalable from WWW: <http://www.microsoft.com/visualstudio/en–us>

[26] Vrakas Dmitris and Vlahavas Ioannis. *ViTAPlan: A Visual Tool for Adaptive Planning*. In Proceedings of the 9th Panhellenic Conference on Informatics. Thessaloniki, Greece. 2003. pp. 167–177.

[27] Oracle Corporation. *Welcome to NetBeans.* [cit. 2012–04–07]. Available from WWW: <http://netbeans.org/>

[28] *XML – Wikipedia, the free encyclopedia.* [online]. Last updated 2012–04–04 [cit. 2012–04–05]. Avalable from WWW: <http://en.wikipedia.org/wiki/XML>

[29] Refsnes Data. *XML Tutorial.* [online]. [cit. 2012–04–05]. Avalable from WWW: <http://www.w3schools.com/xml/>

[30] Plch, T., Chomut, M., Brom, C., Barták, R. *Inspect, Edit and Debug PDDL Documents: Simply and Efficiently with PDDL Studio* : Video presentation. In Proceedings of the 12th International Conference on Automated Planning and Scheduling. Faculty of Mathematics and Physics, Charles University in Prague. 2012.

A List of files

This chapter describes content of the attached CD. The content can be found in table (Tab 1).

Folder	Content
Doxygen	Doxygen generated documentation. Webpage version for web browser in "`html`" subfolder and Rich text file version for printing in "`rtf`" subfolder.
Executables	Microsoft Windows and UNIX versions of PDDL Studio, both logging and standard versions of executable files.
Libraries	Redistributable libraries needed by PDDL Studio to work correctly.
Licence	"`gpl.txt`" file containing licence information
Source	Microsoft Visual Studio 2010 complete project of PDDL Studio. Note that due to usage of Qt framework, Qt add–on must be installed in Microsoft Visual Studio 2010 to work correctly.
Text	Thesis text in "`pdf`" format.
Video	Demonstration videos with description "`pdf`" file [30].

Tab 1 – Content of the attached CD

B User guide

B.1 Requirements

Recommended platform requirements are as follows: Microsoft Windows XP 32 bit or Ubuntu 9.04 32 bit or a compatible platform. Recommended hardware requirements are as follows: Intel dual core 2.26 GHz central processor unit or compatible, 3 GB RAM or compatible, GeForce 9300M GS or compatible video card and 100 MB or more hard drive space. Note that these requirements are "recommended" not minimal, so the program can be expected to run on machines not satisfying these requirements.

B.2 Running the program

This program needs some libraries to run, which libraries it needs depends on program version. Libraries can be found on the attached CD.

B.2.1 MS Windows Version

Windows version of the program requires Qt framework version 4. If Qt framework version 4 is not installed, the program requires Qt framework libraries "`QtCore4.dll`" and "`QtGui4.dll`" stored in the same folder as an executable file. For correct function of icons, the program requires Qt plug–in file "`qico4.dll`" stored in the path "`plugins/imageplugins`" relative to the executable file and empty Qt configuration file "`qt.conf`" in the same folder as the executable file.

The program requires Microsoft Visual C++ Redistributable libraries "`msvcp100.dll`" and "`msvcr100.dll`" either stored in the same folder as executable or as a part of a C++ installation.

B.2.2 UNIX Version

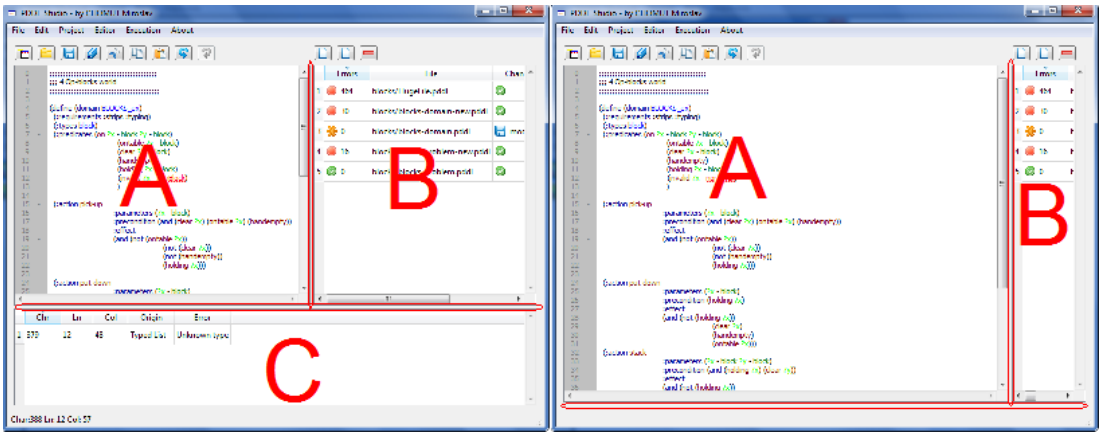
The UNIX version of the program requires Qt framework version 4.7. If Qt framework version 4 is not installed, the program requires Qt framework libraries "`libQtCore.so.4.7.3`" and "`libQtGui.so.4.7.3`" stored in "`/lib`" folder together with links to them "`libQtCore.so.4.7`" and "`libQtGui.so.4.7`". For correct function of icons, the program requires Qt plug–in file "`libqico.so`" stored in the path "`plugins/imageplugins`" relative to the executable file and empty Qt configuration file "`qt.conf`" in the same folder as the executable.

B.3 Demo video

Attached CD contains video presentation along with "`video_desc.pdf`" description file [30]. It was submitted as demonstration on the 12th International Conference on Automated Planning and Scheduling. The video presentation demonstrates application's basic features.

B.4 Main window

When the program starts, an empty main window with no project is shown. Main window consists of these parts: menu, quick buttons, source code editor (Pic B.1 (A)), list of project's files (Pic B.1 (B)), Interactive Error Table (Pic B.1 (C)) and status bar. Main parts of this window can be repositioned to better match developer's needs. Repositioning is made by two reposition sliders (highlighted part in picture Pic B.1). Each part has its minimal size, however sliders can be moved to far edge which results in certain part being completely invisible and slider remains in that edge. Example of window before and after reposition can be seen on a picture (Pic B.1).

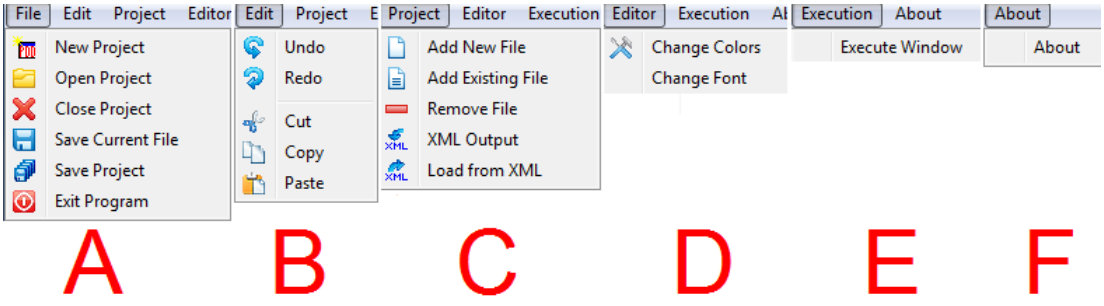


Pic B.1 – Main window, source code editor (A), list of project's files (B), Interactive Error Table (C), sliders (highlighted part)

B.4.1 Menu

Menu is in the upper part of the window (Pic B.2) and it is divided into six basic parts: "`File`", "`Edit`", "`Project`", "`Editor`", "`Execution`" and "`About`".

Pic B.2 – Menu is located in top part of the application window



Pic B.3 – All menus and their options

These are actions available from individual menus:

- From the part "`File`" (Pic B.3 (A)), a developer can manage a project as a whole. Possible actions are: creating a new project, opening a project, closing a project, saving an active file, saving all files and closing the program.
- From the part "`Edit`" (Pic B.3 (B)), a developer can perform basic tasks with the source code editor. Actions consists of: undoing last change in the text, redoing it, cutting or copying selected text and pasting text from the clipboard.
- From the part "`Project`" (Pic B.3 (C)), a developer can manage individual project's files. Possible actions are: creating a new file, adding an existing file, removing a file and managing XML import and export feature.
- From the part Editor (Pic B.3 (D)), a developer can change the source code editor's appearance. Possible actions are changing font or opening settings window for highlights color change.
- From the part "`Execution`" (Pic B.3 (E)), a developer can open a window handling third party software integration. From which he/she can execute third party planners.
- From the part "`About`" (Pic B.3 (F)), a developer can open the "about" window to see some program details.

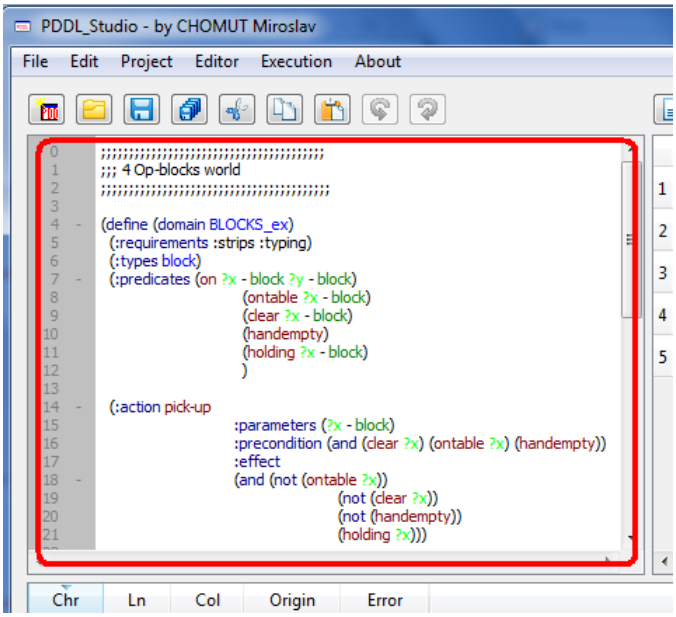
B.4.2 Quick buttons

Below menu are quick buttons (Pic B.4), which provide a developer with shortcuts to their functions. Quick buttons are differentiated into two sections so they are placed closer to the part they affect.

Pic B.4 – Quick buttons are positioned on the top part of the application window. Quick buttons related to project's files list are positioned on the right while others are on the left.

B.4.3 Source code editor

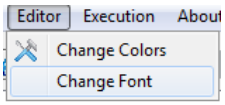
Source code editor resides in the left part of the window (Pic B.5). This part of the window is expected to be most frequently used. It allows a developer to actually change the code of the PDDL files. Source code editor is equipped with standard features like scrollbars for orientation in bigger files and some advanced features like Line Counter, Code Highlighting, Code Collapsing and Code Completion.



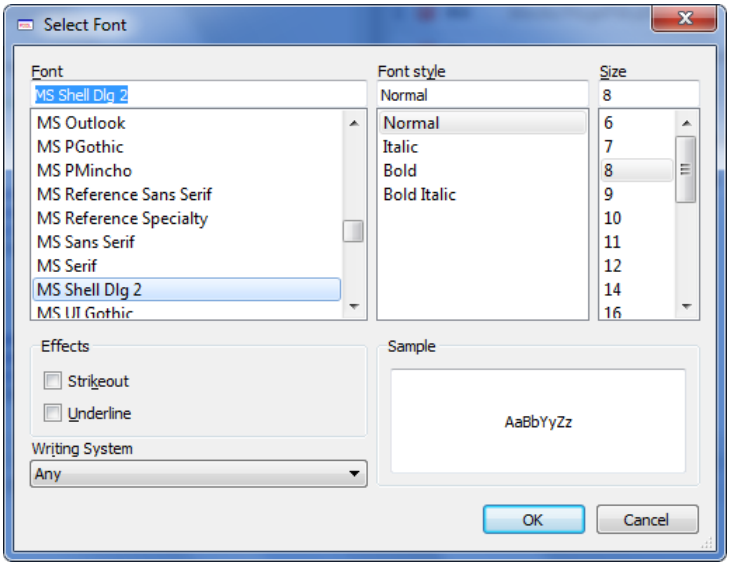
Pic B.5 – Source code editor

B.4.3.1 Font change

Developer can change the source code editor's font from the menu "`Editor / Change Font`" (Pic B.6). Font change dialog appears (Pic B.7), font dialog is dependent on your system, but could look similar to our example. After selecting a new font, the font of the source code editor is changed.



Pic B.6 – Change Font menu field



Pic B.7 – System font select window (may differ from platform to platform)

B.4.3.2 Line Counter

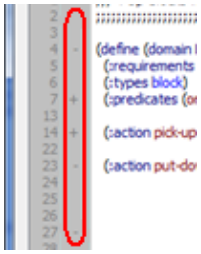
For better orientation in the code, the left part of the source code editor contains column reserved for Line Counter (Pic B.8). Line counting starts from the zero. This feature is mostly helpful when many parts of the code are collapsed.

Pic B.8 – Line Counter, lines 7 and 14 contains collapsed blocks

B.4.3.3 Code Collapsing

Code Collapsing regions are automatically determined during the parsing therefore it is advised to let the text parsing proceed to correct regions. Code Collapsing switches are in the left column right next to the Line Counter (Pic B.9).

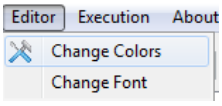
Code Collapsing switches are represented by "+", which represents collapsed data that can be shown, and by "−", which represents start of the region that can be collapsed. Collapsing switches are operated by a simple mouse click.



Pic B.9 – Code Collapsing switches

B.4.3.4 Syntax Highlighting

Text in the source code editor is highlighted for better orientation. Highlighting takes place after parsing, therefore parsing process is needed for highlighting to match with actual text. Highlighting colors can be changed in a settings window, which is opened from the menu "`Editor / Change Colors`" (Pic B.10).

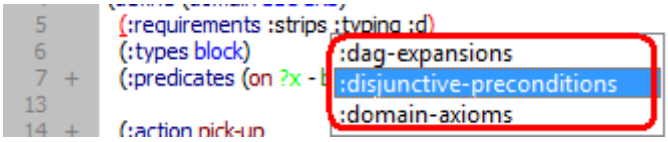


Pic B.10 – Change Colors menu field

B.4.3.5 Code Completion

When a developer writes his/her creation, the Code Completion provides him/her with contextual hints and it can complete the actual word. Code Completion is dependent on the parsed text, therefore for most up–to–date completion context, it is advised to let the parsing process to run from time to time. Completion possibilities

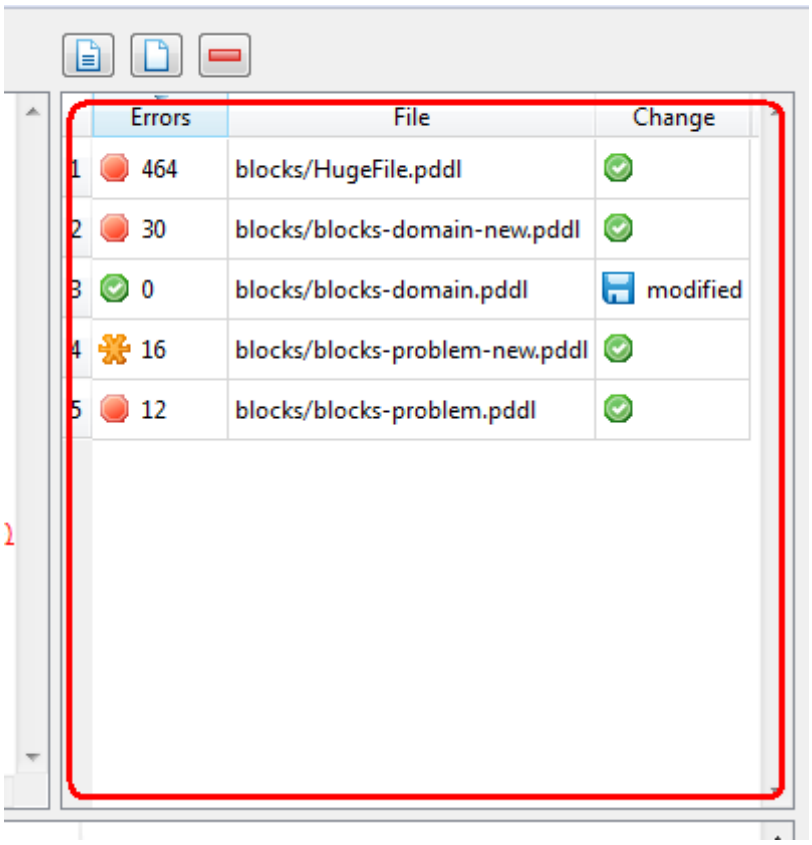
appear in popup window (Pic B.11), navigation in window can be made by mouse or cursor keys. Completion is accepted by "`Return`" key and by "`Esc`", "`Alt`" or "`Ctrl`" key, the completion popup is closed.



Pic B.11 – Code Completion list, second option is selected

B.4.4 List of project's files

In the right part of the window resides a list of project's files (Pic B.12). List itself has three columns. First column informs about whether a certain file contains an error or is actually edited. First column also contains a number presenting an error count per file. Error count is refreshed once per "`allErrCheck`" time, which can be set via a configuration file. Second column contains a file relative path with its name. The last column informs whether the file is modified from the original on the persistent storage. File selection is made by double–clicking the row with the desired file.

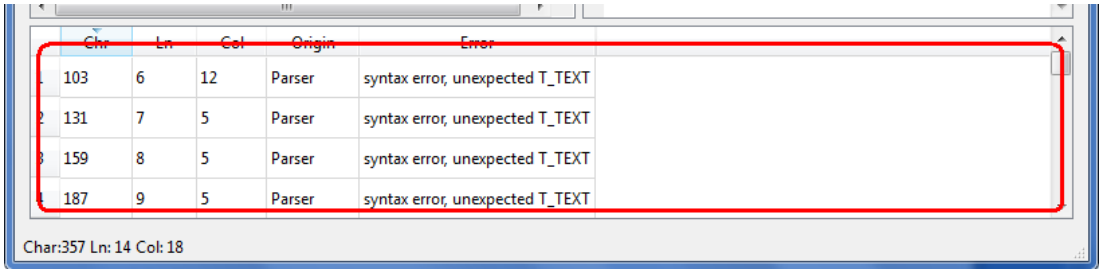


Pic B.12 – List of project's files

B.4.5 Interactive Error Table

At the bottom part of the window resides an Interactive Error Table (Pic B.13). Error table consists of five columns: first column denotes the error position, second column denotes the line of the detected error, third column denotes the column of the detected error, fourth column informs about the origin of the error and the last

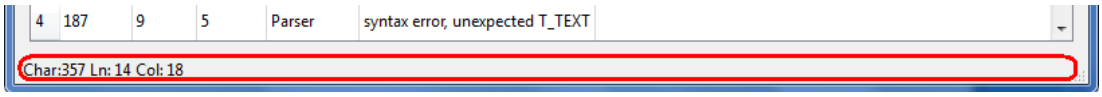
column describes the error itself. Developer can sort this table by any column alphabetically by clicking on its header. When a developer double–clicks on any error, the text cursor in the source code editor is set to the corresponding position.



Pic B.13 – Interactive Error Table

B.4.6 Status bar

At the bottom of the window is a status bar (Pic B.14), which informs a developer about exact location in the source code editor. Status bar also informs developer when the parsing process is in effect "`---working---`", all files are being checked for errors "`---global error check---`" or Auto–Save feature is storing changes on persistent storage "`---safety saving---`".



Pic B.14 – Status bar

B.4.7 Project management

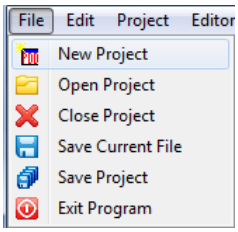
Program works with so called projects. Each project consists of two file types, one project file with preferred extension "`pddl_proj`" and none or more PDDL files with preferred extension "`pddl`". Project files store information about PDDL files.

Project management consists of creating, opening and closing projects, creating, adding and removing PDDL files and saving files. Also selecting active file from project's files is part of the Project Management.

B.4.7.1 Creating new project

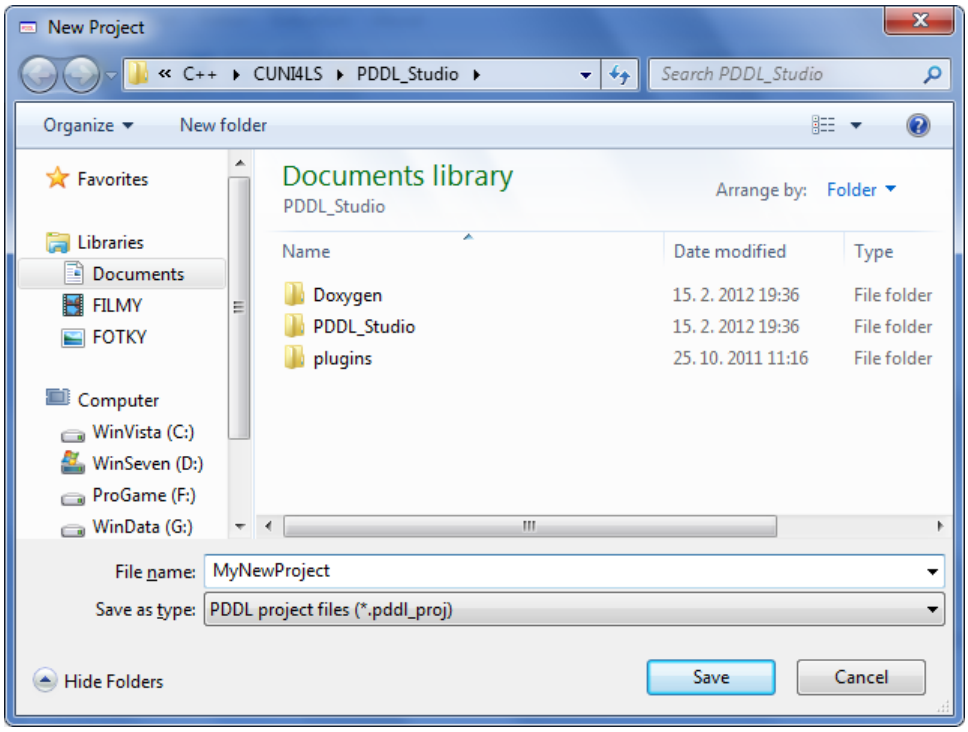
Upon creating a new project, the program closes the previously opened project if any and creates a new one. The new project is created without any PDDL files. From the main window a developer can create the new empty project with "`Create new project`" (Pic B.15) quick button or from the menu "`File/New Project`" (Pic B.16).

Pic B.15 – Create new project quick button



Pic B.16 – New Project menu field

File select dialog appears (Pic B.17), the file dialog is dependent on your system, but could look similar to our example. After selecting the file name, new empty project is created.

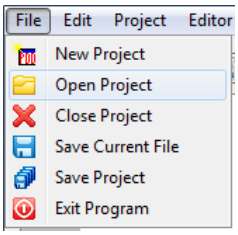


Pic B.17 – File select dialog (may differ from platform to platform)

B.4.7.2 Opening existing project

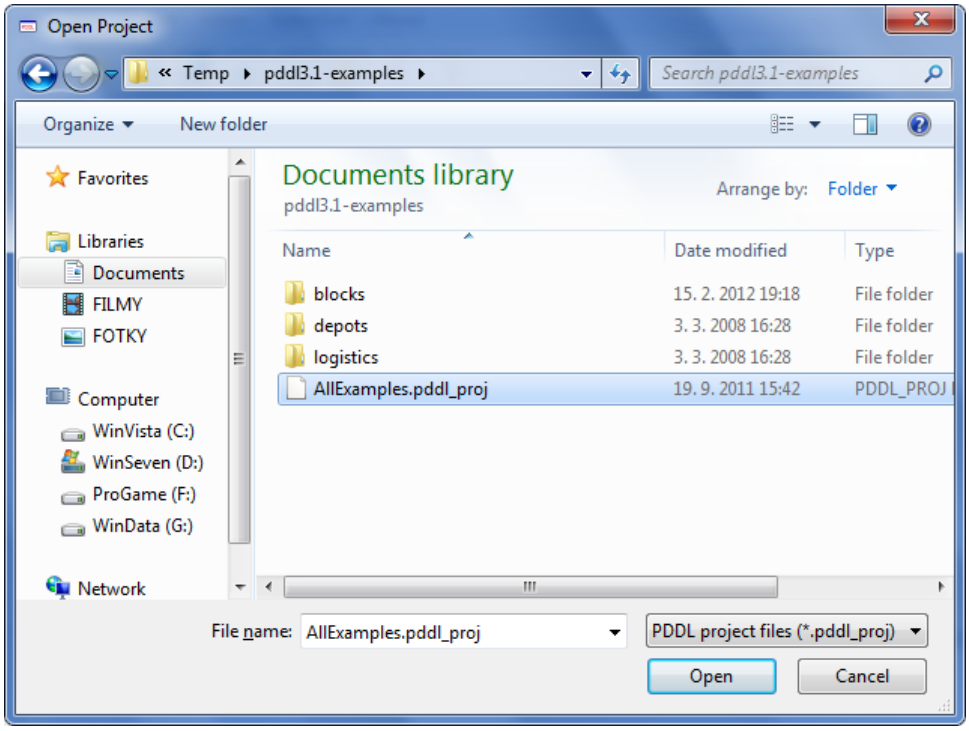
Upon opening a project, the program closes previously opened one if any and loads the new one. After loading the project, all files are parsed and global error detection is executed. From the main window a developer can open an existing project with "`Open project`" (Pic B.18) quick button or from the menu "`File/Open Project`" (Pic B.19).

Pic B.18 – Open project quick button



Pic B.19 – Open project menu field

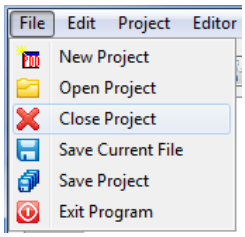
File select dialog appears (Pic B.20), the file dialog is dependent on your system, but could look similar to our example. After selecting the file name, existing project is opened.



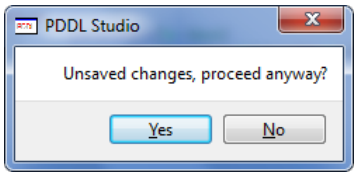
Pic B.20 – File select dialog (may differ from platform to platform)

B.4.7.3 Closing current project

Currently opened project will be closed whenever a new project is created, other project is opened, the program is closed or from "`Menu/Close Project`" (Pic B.21). If any unsaved changes are present in the project during its closing, a developer is asked whether he/she really wants to proceed with the operation (Pic B.22).



Pic B.21 – Close Project menu field

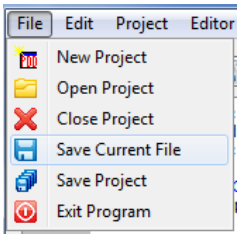


Pic B.22 – Confirmation message box

B.4.7.4 Saving active file

When a developer wants to save PDDL file currently being edited, it can be done by "`Save current file`" (Pic B.23) quick button or from the menu "`File/Save Current File`" (Pic B.24).

Pic B.23 – Save current file quick button

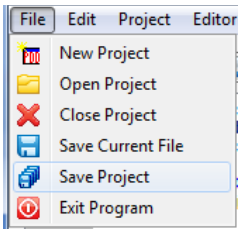


Pic B.24 – Save Current File menu field

B.4.7.5 Saving full project

When a developer wants to save all PDDL files and a project file, it can be done by "`Save all files and project file`" (Pic B.25) quick button or from the menu "`File/Save Project`" (Pic B.26).

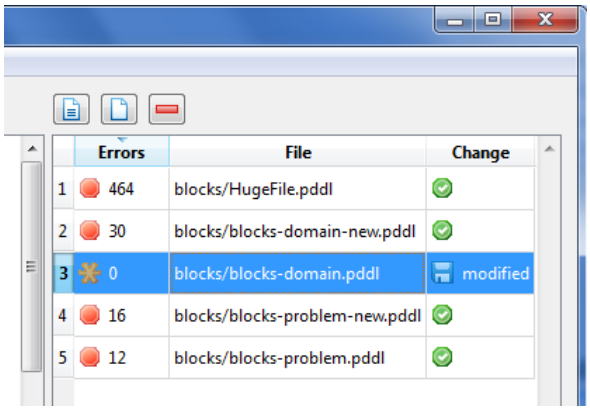
Pic B.25 – Save all files and project file quick button



Pic B.26 – Save Project menu field

B.4.7.6 Selecting an active file

Active PDDL file, the one which is being edited, is selected by double–click on the file name in the list of project's files (Pic B.27). When the file is selected, its content is loaded into the source code editor and its status in the list is set to active.

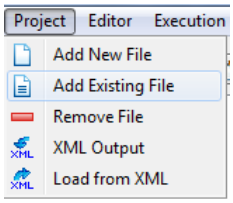


Pic B.27 – Selecting file by double–click, active file has star instead of error status

B.4.7.7 Add existing file to project

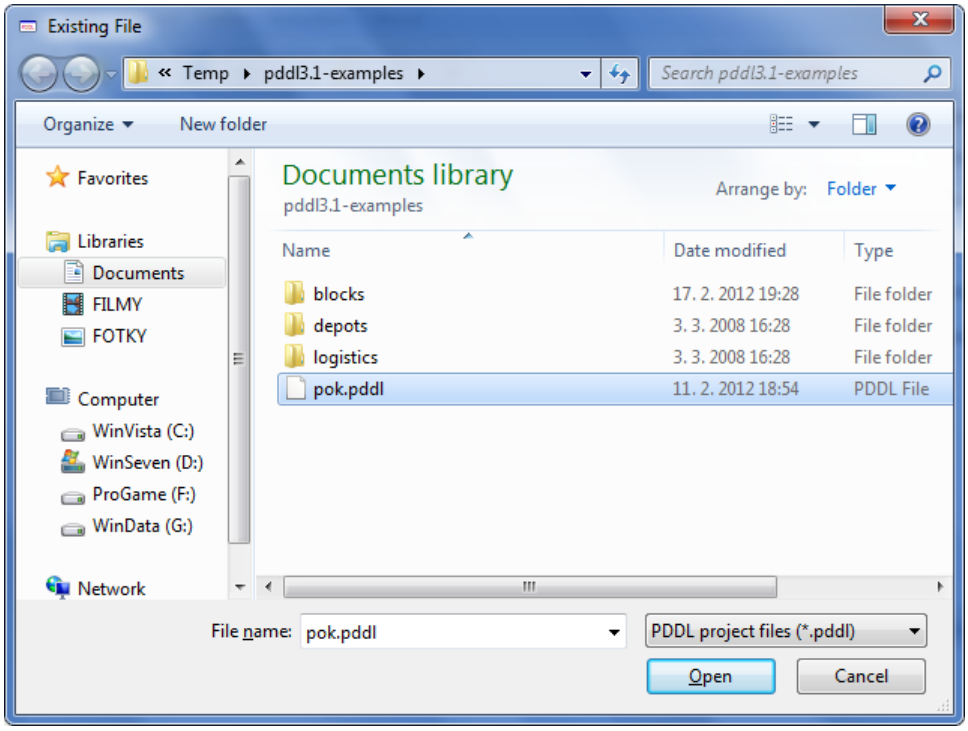
When a developer wants to add an already existing file into the project, he/she can do this by "`Add existing file into project`" (Pic B.28) quick button or from the menu "`Project / Add Existing File`" (Pic B.29).

Pic B.28 – Add existing file into project quick button



Pic B.29 – Add Existing File menu field

File select dialog appears (Pic B.30), the file dialog is dependent on your system, but could look similar to our example. After selecting the file name, the existing file is added into the project. The project must be saved for change to be persistent.

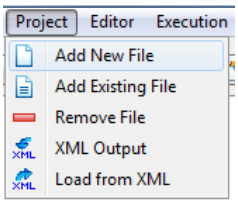


Pic B.30 – File select dialog (may differ from platform to platform)

B.4.7.8 Add new file to project

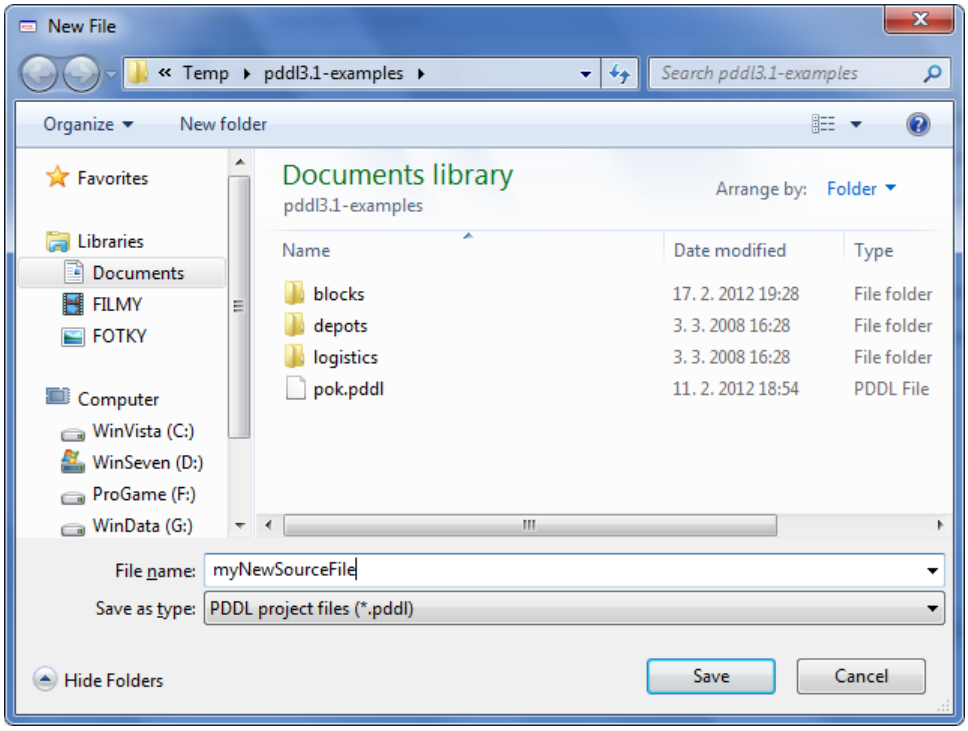
When a developer wants to create a new file in the project, he/she can do this by "`Add new file into project`" (Pic B.31) quick button or from the menu "`Project / Add New File`" (Pic B.32).

Pic B.31 – Add new file into project quick button



Pic B.32 – Add New File menu field

File select dialog appears (Pic B.33), the file dialog is dependent on your system, but could look similar to our example. After selecting a file name, new file is created and added into the project. The project must be saved for the change to be persistent.

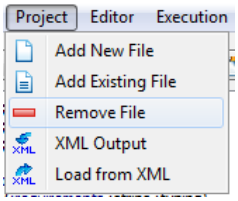


Pic B.33 – File select dialog (may differ from platform to platform)

B.4.7.9 Remove file from the project

When a developer wants to exclude a file from the project, he/she can do so by selecting that file as actual and utilize "`Remove file from project`" quick button (Pic B.34) or from the menu "`Project / Remove File`" (Pic B.35).

Pic B.34 – Remove file from project quick button



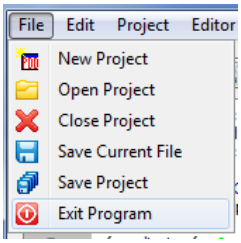
Pic B.35 – Remove File menu field

The file must be saved prior to being removed from the project. When the file is removed from the project, it remains intact on the hard drive and the project needs to be saved for the change to be persistent.

B.4.8 Program termination

The program can be terminated from the main window by the closing button (Pic B.36) or from the menu "`File/Exit Program`" (Pic B.37).

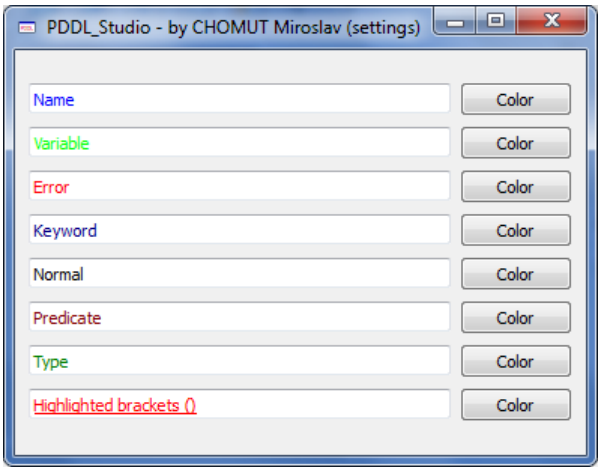
Pic B.36 – Teminate button



Pic B.37 – Exit Program menu field

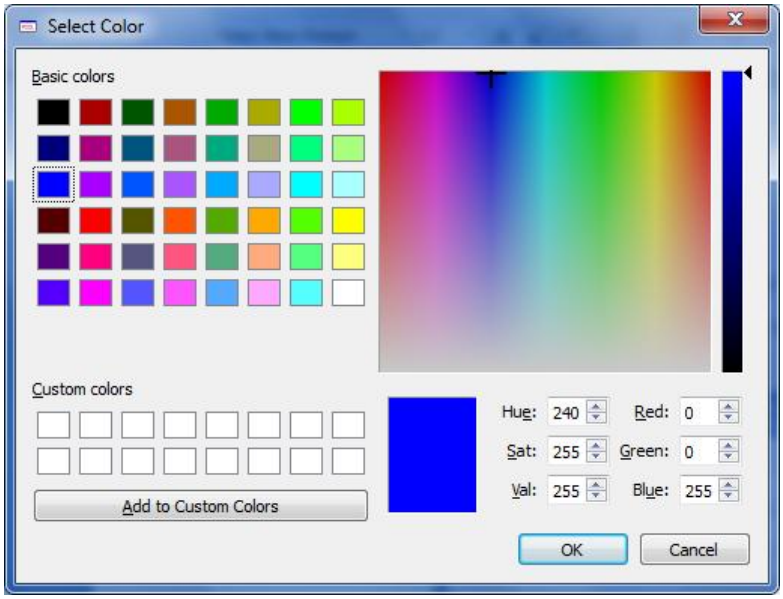
B.5 Settings window

The settings window (Pic B.38) is used for change of highlight colors. Left part of the window contains highlight types with preview and the right part contains buttons, which are used for changing of highlight colors.



Pic B.38 – Settings window

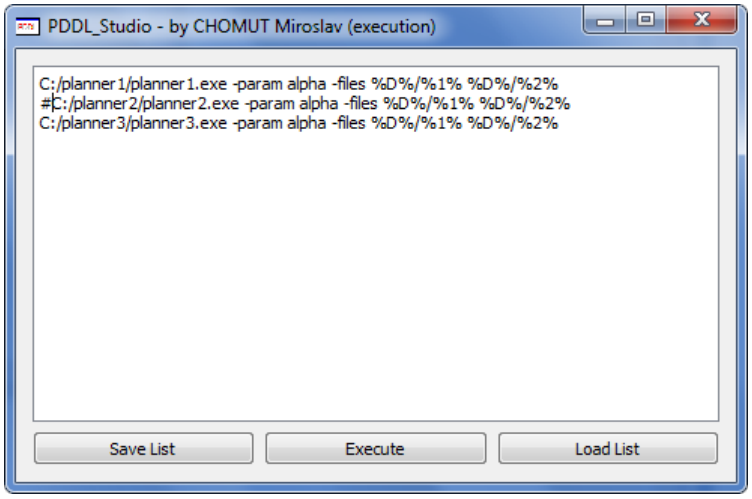
When a button is clicked the color select dialog appears (Pic B.39), color dialog is dependent on your system, but could look similar to our example. After selecting the color, the affected highlight is changed. Highlighting in the source code editor updates on the next parsing process.



Pic B.39 – Color select dialog (may differ from platform to platform)

B.6 Executor window

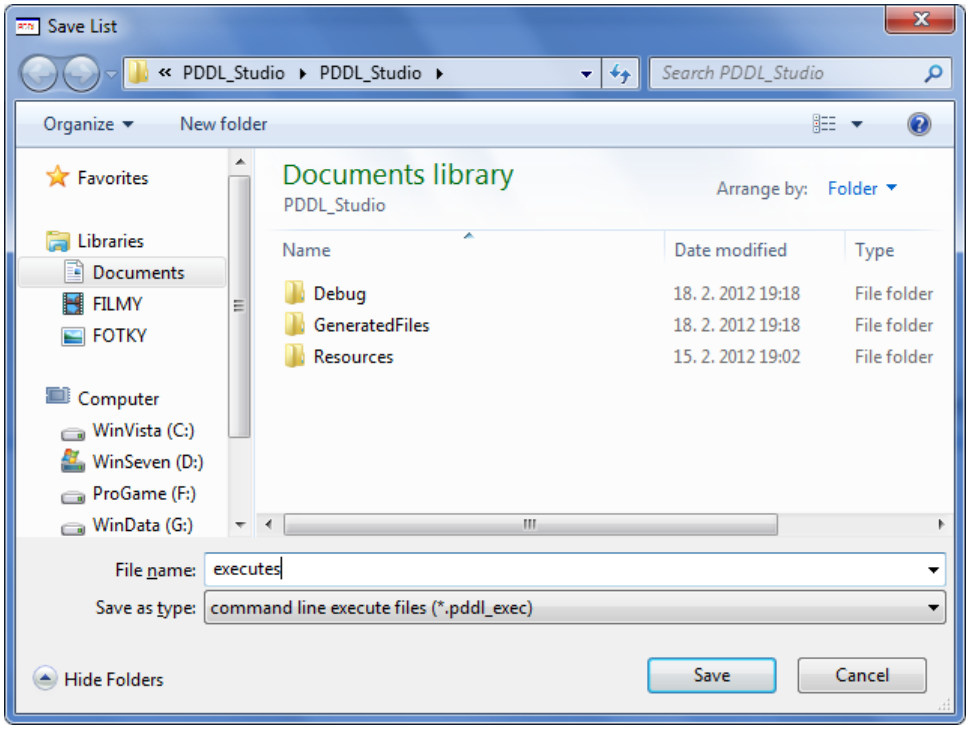
From the executor window (Pic B.40) a developer can run various batch programs, for example multiple planners to compare their performance.



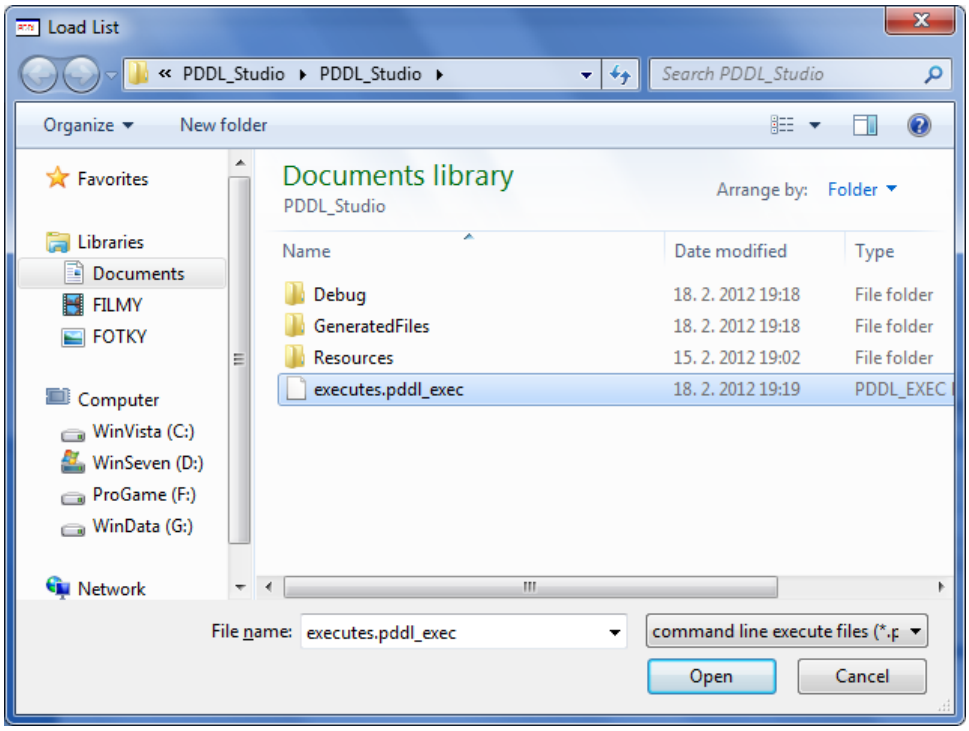
Pic B.40 – Executor window

Upper part of the executor window contains commands. Each line is treated as one command. Lines prefixed with "`#`" are ignored. Special variables are translated to the appropriate text. Variable "`%D%`" is translated to the project's directory path and variables "`%0%`" to "`%n%`" where n is number of project's files are translated to project's file names. Otherwise commands are passed to system as are. Commands are executed by "`Execute`" button.

Command can be saved by "`Save List`" button or loaded when previously saved by "`Load List`" button. Save file dialog (Pic B.41) and Load file dialog (Pic B.42) are dependent on your system, but could look similar to our example. File for saved execution commands has extension "`pddl_exec`".



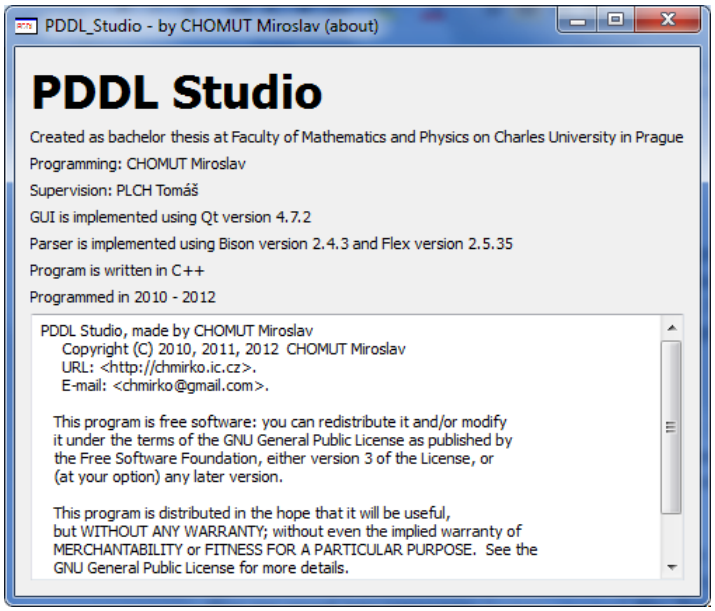
Pic B.41 – Save file dialog (may differ from platform to platform)



Pic B.42 – Load file dialog (may differ from platform to platform)

B.7 About window

Window about (Pic B.43) shows some information about PDDL Studio creation and licensing.



Pic B.43 – About window

B.8 Configuration file

PDDL Studio has some features which are configurable by a configuration file. The configuration file is stored as "`pddl.conf`" in the directory of the binary file. If no configuration file is present, default one is created. One line in the file represents one setting. Each line starts by a variable name and after a space there is a value of that variable. Unknown lines are skipped.

B.8.1 Parser time

Reaction based parsing is configurable within this file. Variable "`reaction`" represents time in milliseconds during which a developer must not do any change for parsing process to kick in. Default value is 500 milliseconds.

B.8.2 Auto–Save feature

Auto–Save feature is configurable within this file. Variable "`safetyTime`" represents time in milliseconds. Each "`safetyTime`" milliseconds, all modified files are saved into the temporary "`pddl_bckp`" file. Temporary files are erased on file's save or correct project's close. When application from any reason crashes, on the next project's open, temporary files are loaded as modified content of project's files. Default value is 60000 milliseconds (1 minute).

B.8.3 Global error check

Global error check timing is configurable within this file. Variable "`allErrCheck`" represents time in milliseconds. Each "`allErrCheck`" milliseconds all project's files are checked for errors and result from this operation is shown in the list of project's files. Default value is 20000 milliseconds (20 seconds).